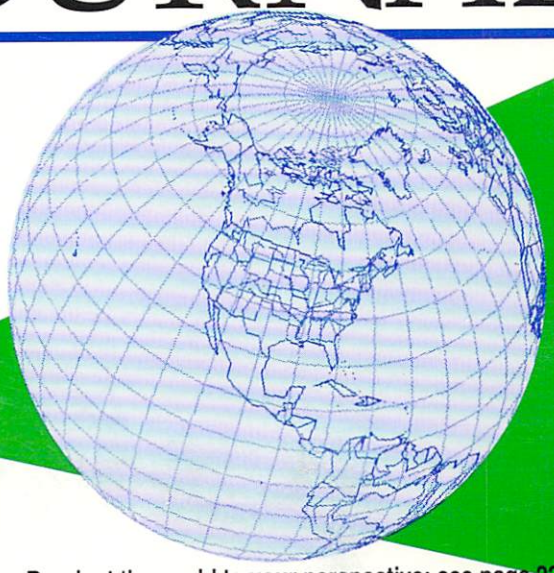
Disk Inside

source code and executables for all articles

AMIGA WORLD

TECH JOURNAL

ARTICLES



Reorient the world to your perspective; see page 22.

REVIEWS

COLUMNS

ON DISK

(See page 33.)

A68k version 2.71: ***A full-featured assembler, FREE***

BLINK version 6.7: ***THE linker to use***

Plus source code and executables for articles

0 74470 12070 4 10

U.S.A. $15.95
Canada $19.95

Volume 1, Number 4 | October 1991 | An IDG Communications Publication

SAVE IT. MOVE IT. GET IT BACK.

Back-Up...Transfer...Retrieve Quickly And Easily With Central Coast's Software For The Amiga

Valuable utility programs can save you time, money and, in the case of catastrophic errors like hard drive failure, possibly months of work.

Quarterback Tools – Recover Lost Files

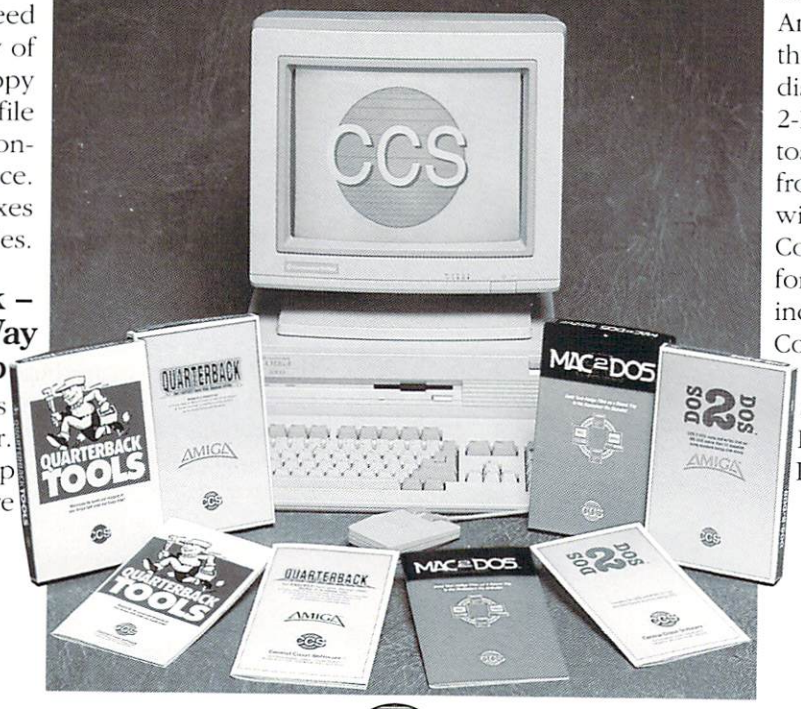
Fast and easy. Reformats all types of disks – either new or old filing systems – new or old Workbench versions. Also optimizes the speed and reliability of both hard and floppy disks. Eliminates file fragmentation. Consolidates disk space. Finds and fixes corrupted directories.

Quarterback – The Fastest Way To Back-Up

Backing-up has never been easier. Or faster. Back-up to, or restore from: floppy disks, streaming tape (AmigaDOS-compatible), Inner-Connection's Bernoulli drive, or ANY AmigaDOS-compatible device.

Mac-2-Dos & Dos-2-Dos – A Moving Experience

It's easy. Transfer MS-DOS and ATARI ST text and data files to-and-from AmigaDOS using the Amiga's own disk drive with Dos-2-Dos; and Macintosh files to-and-from your Amiga with Mac-2-Dos. Conversion options for Mac-2-Dos include ACSII, No Conversion, MacBinary, Postscript, and MacPaint to-and-from IFF file format.



CCS

Central Coast Software
A Division Of New Horizons Software, Inc.

206 Wild Basin Road, Suite 109, Austin, Texas 78746
(512) 328-6650 * Fax (512) 328-1925

Quarterback Tools, Quarterback, Dos-2-Dos and Mac-2-Dos are all trademarks of New Horizons Software, Inc.

Circle 3 on Reader Service card.

The AmigaWorld Tech Journal (ISSN 1054-4631) is an indepedent journal not connected with Commodore Business Machines, Inc. *The AmigaWorld Tech Journal* is published bimonthly by IDG Communications/Peterborough, Inc., 80 Elm St., Peterborough, NH 03458. U.S. Subscription rate is $69.95, Canada and Mexico $79.95, Foreign Surface $89.95, Foreign Airmail $109.95. All prices are for one year. Prepayment is required in U.S. funds drawn on a U.S. bank. Application to mail at 2nd class postage rates is pending at Peterborough, NH and additional mailing offices. Phone: 603-924-0100. **Postmaster:** Send address changes to *The AmigaWorld Tech Journal*, 80 Elm St., Peterborough, NH 03458. *The AmigaWorld Tech Journal* makes every effort to assure the accuracy of articles, listings and circuits published in the magazine. *The AmigaWorld Tech Journal* assumes no responsibility for damages due to errors or omissions. Third class mail enclosed.

Bulk Rate
US Postage Paid
IDG Communications/Peterborough, Inc.

MESSAGE PORT

"Working" doesn't mean "right."

"It's weird, but it works."

"It'll do for now, I'll clean it up in the next revision."

"No one's ever going to use it but me, what does it matter?"

Which is your excuse for breaking the programming rules? They all sound fine when you're frustrated. The problems arise when you alter your system and "it" doesn't work, your angry customers report bugs, friends start copying your bad technique, and Commodore upgrades the operating system.

While plenty of fun has been had at the expense of Commodore over this latest upgrade ("Which takes longer finishing 2.0 or teaching your cat to speak Klingon?"), we all share much of the blame for the delay. For years programmers have been taking short cuts and capitalizing on quirks and bugs in 1.3 instead of taking the time to do things the sanctioned way. Sending out early versions of 2.0 to developers gave CBM a rude surprise. With many of the OS' bugs fixed, many existing (and some respected) programs broke under the new version. Much of the 2.0 development team's time was subsequently consumed trying to make 2.0 backward compatible, with new problems popping up continually. Some features and fixes even had to be scrapped because they conflicted with old-favorite cheats. If everyone had followed Commodore's programming rules from the start, we wouldn't have had such a long wait for the 2.0 upgrade.

While the rules have always been available—in the ROM Kernel manuals, DevCon notes, and *AmigaMail*—you often had to root them out. With little enforcement, there was no great incentive to follow them. The situation is now changing. Aggressively pushing compatibility with 2.0's new look, Commodore has compiled the *Amiga User Interface Style Guide*, which clearly lays down the CATS-approved law. Adding support is the third edition of the *AmigaDOS Manual*. New ROM Kernel manuals will complete the picture.

The AmigaWorld Tech Journal is doing its part too. Our authors and Peer Review Board are dedicated to teaching, not just the latest techniques but, more importantly, the proper techniques and new 2.0 standards. What good is getting a project done quickly if it breaks equally quickly while on a different system, multitasking, or under a new version of the OS? If you start programming by the rules now, you'll be ready for the next generation of improvements Commodore has planned. You can bet they won't be as forgiving as 2.0.

Speaking of doing things right, our apologies to Commodore. Our June/July '91 issue carried contradictory messages on the distribution rights for the 1.3 include files. Reading the text on page 25 leads you to believe the includes are freely distributable. Reading the copyright statement on the disk tells you that the files are not and that Commodore reserves all rights. Yours truly didn't proofread carefully enough. **Commodore's 1.3 and 2.0 include files are only distributable under license from Commodore Business Machines**. Sorry about any confusion this caused.

The Complete Guide for the Blittering Idiot

How (and when) to control the Blitter.

By Tomas Rokicki

THE CUSTOM CHIP set that forms the basis of the Amiga's architecture is both a great strength and a severe handicap. These custom chips and the capabilities they provide—fast, flashy graphics, stereo sound, and astonishing video tricks—distinguish the Amiga from its competition. But, as processors and memory get faster and display resolutions increase, the limitations of these chips—which are the Amiga's very foundation—become more apparent. As a programmer, you must know not only how to take advantage of the custom chips, but also when it is appropriate to do so, and when it is not.

The Blitter is one of the more complicated functions of the Amiga custom chip set. When properly used, it can provide significant improvements in performance in many different ways. This article provides an introduction to programming the Blitter; after reading it, you will be able to use the Blitter to perform a number of typical graphics operations. The primary motivating example, and indeed, the primary purpose of the Blitter, will be the movement of an arbitrary rectangle of bits from one location to another on a screen.

Before you dive in, be warned: You should have a basic familiarity with the graphics architecture of the Amiga—how bitplanes are stored in memory, how the bitplanes are combined to form screens with multiple colors, and how this memory is addressed. A code fragment such as

```
void setbit(short *bitplane, int width, int x, int y) {
    bitplane[y * width + (x >> 4)] != (0x8000L >> (x & 15)) ;
}
```

should be understandable without more than a few minutes' thought. You should also be prepared to spend some time with this article; some of the difficulties in programming the Blitter can be subtle and technical.

Understanding how the Blitter works is essential even if you do not plan to access it directly; understanding how the graphics library uses the Blitter can help you write faster code. Even more important, you can avoid subtle bugs that may appear only on newer, faster machines or under strange display circumstances.

BLITS FOR BEGINNERS

The Blitter is like one of those kitchen aids you see advertised on late night television—it can do a lot of things. It can copy areas of a screen—when you move a window on the Amiga, the Blitter copies the window graphics to their new position. It can clear or set any rectangle on the screen—the borders of windows are drawn with this Blitter function. It can perform complex logical operations to merge data from multiple sources, such as filtering a graphics image through a stencil. It can also shift pixel images horizontally or vertically. While we will not discuss this aspect, the Blitter can fill arbitrary enclosed areas in several different ways and draw lines at any angle and in several different modes as well. All figures and descriptions from here on will ignore the Blitter's line-mode capabilities, but watch for a future article on line-mode and the area fill capabilities.

Much of the speed advantage in using the Blitter comes from the fact that it runs in parallel with the Amiga's main processor. Once the Blitter is told what to do, it performs that operation concurrently with the processor. Most of the system software's graphics routines use this feature. For example, executing Draw() programs the Blitter to draw the line and start it up; the actual Draw() call returns before the line is complete. This way the processor can compute the endpoints of the next line as the Blitter draws the current one.

An important fact is that the graphics operation may not have completed before the subroutine call returns if the processor will reference the same memory. For instance, if you clear a region of memory with the Blitter in preparation for rendering into that memory with the processor, you must make sure that the Blitter operation has completed before starting any rendering. The graphics library call WaitBlit() will perform this check; WaitBlit() will not return until the current Blitter operation has completed.

For example, let us say that we want to use the setbit() routine to turn on a certain number of pixels on a screen, and that we clear that screen with the graphics operation RectFill(). The following code will fail:

```
RectFill(rp, minx, miny, maxx, maxy) ;
for (i=0; i<1000; i++)
    setbit(rp->BitMap->Planes[0], rp->BitMap->BytesPerRow/2,
        randm(minx, maxx), randm(miny, maxy)) ;
```

Even if all of the appropriate auxiliary functions exist, the code will fail because the RectFill() call returns before the Blitter finishes the operation. The processor will set some bits, but many of them will be cleared as the Blitter proceeds with its operation.

Adding a call to WaitBlit() immediately after the RectFill() procedure solves this problem. Note that WaitBlit() in this case is conservative; if there is a task switch after RectFill(), and another task starts a Blitter operation, then the WaitBlit() will wait until that task's Blitter call is complete. Thus, if you start a short blit and call WaitBlit(), there is no guarantee that the WaitBlit() call will return quickly. This type of thing happens with sufficient rarity that it may be, for the most part, ignored.

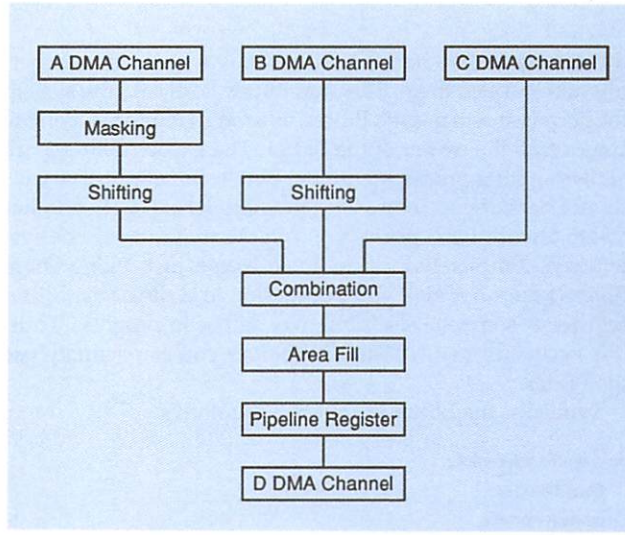


Figure 1: Blitter data path block diagram.

Take advantage of the parallelism between the Blitter and the processor whenever possible. For example, if a large section of memory needs to be cleared and some extensive calculations need to be performed, it is advantageous to perform the memory clear with the Blitter, starting the operation right before performing all of those calculations. Thus,

```
RectFill(rp, minx, miny, maxx, maxy) ;
WaitBlit() ;
DoFlightCalculations() ;
```

is not an optimal use of the Amiga. Moving the WaitBlit() call to after DoFlightCalculations() will improve the code's performance.

The Blitter has one primary limitation—it can only access chip memory. This is the lower region of memory used for all displayed graphics; in current Amiga models it is the lowest half to two megabytes of memory. Attempting to use the Blitter to access other memory will cause severe memory trashing. If you allocate memory with the intention that it be used by the Blitter, you should set the MEMF_CHIP flag in the allocation request:

```
void *getChipRam(long size) {
   return AllocMem(size, MEMF_CHIP) ;
}
```

THE PROCESSOR'S PERSPECTIVE

While the Blitter may be controlled by the processor or the Copper, we shall only consider programming it from the processor. In either case, the Blitter appears as a set of write-only registers and a single read-only register in the custom chip address space. To access these registers from C, you should include the file hardware/custom.h in your program. Unfortunately, depending on your compiler's version and the model you are compiling with, accessing the custom registers may require some additional code. Normally, you access the custom registers with the code below:

```
#include <exec/types.h>
#include <exec/hardware/custom.h>
extern struct Custom custom ;
```

Under SAS/C 5.10 with the small data model, you must add the following lines:

```
struct Custom *custom_ptr = &custom ;
#define custom (*custom_ptr)
```

Under the small data model, all data accesses are made as a 16-bit offset of an address register, and the value of this address register is determined when the program is loaded. The custom hardware registers, on the other hand, are at a fixed location in memory that is not within a 16-bit offset of any data memory. Adding the above lines allocates and initializes a pointer to the custom hardware registers in the normal data area and then redefines custom to use this pointer rather than the external symbol.

Other compilers will require other tricks. For instance, the Manx Aztec 3.6a C compiler comes with include files that perform a trick similar to the one above—but using this include file prohibits you from using the line:

```
extern struct Custom custom ;
```

So this is somewhat of a mess. I do the following, which works under all C compilers and all data models I've tested:

```
#include <exec/types.h>
#include <hardware/custom.h>
#ifdef custom
#undef custom
#endif

#define custom (*(struct Custom *)(0xdff000L))
```

While this is somewhat ugly, it works without complaints or warnings on all of the C compilers I've tried.

To program the Blitter, you simply write the appropriate values into all of the Blitter registers. The complete list of registers and their types is:

```
APTR  bltapt, bltbpt, bltcpt, bltdpt ;
UWORD  bltamod, bltbmod, bltcmod, bltdmod ;
UWORD  bltadat, bltbdat, bltcdat ;
UWORD  bltcon0, bltcon1 ;
UWORD  bltafwm, bltalwm ;
UWORD  bltsize ;
```

(Note that they are not organized this way in memory.)

Each register is referred to by prefixing "custom." to its name. For example, you would refer to the bltsize register as custom.bltsize in C. All of the Blitter registers are write-only. You must never read these locations, and you must make sure to set any unused bits to zero. Most important, you must always write the bltsize register last, because writing to this register starts the Blitter operation.

Before we get further into the details of the contents of ►

these registers, we have to discuss how a program gets permission to touch them. The OwnBlitter() call asks the system for permission to use the Blitter; as soon as it returns, you are considered the owner of the Blitter. The DisownBlitter() call allows another process to use the Blitter. Be careful: The calls do not nest; if you follow an OwnBlitter() call with another call to OwnBlitter() before you call DisownBlitter(), the system will completely lock up. Even worse, just calling OwnBlitter() does not give you permission to write to the Blitter registers—someone else's blit may still be in progress. Thus, it is necessary to call WaitBlit() before you can actually use the Blitter.

Typically, the Blitter is accessed as follows:

```
void rendership(void) {
    OwnBlitter() ;
    while (! done) {
        ... // perform calculations for this blit
        WaitBlit() ;
        ... // write to the custom registers and start the blit
    }
    DisownBlitter() ;
}
```

The Blitter must not be owned for an excessive length of time, as many system routines use it. Almost all rendering to the screen is done with the Blitter; this includes scrolling, window movements, text rendering, even gadget highlighting. Under AmigaDOS 1.3 and earlier, floppy disk data encoding and decoding is done with the Blitter. As a rule of thumb, try to keep the Blitter for less than a small fraction of a second.

The code above shows exactly how almost all of the graphics routines in graphics.library work. The WaitBlit() should be called at the latest possible moment, but before you touch any of the Blitter registers, and the subroutine should return even as the last blit is completing.

Be very careful about what you call between OwnBlitter() and DisownBlitter(); if any of the routines you call allocate the Blitter themselves, the system will lock up. For instance, you should not printf() any debugging statements to a window inside these two calls, as the Text() call that renders text itself requires the Blitter. You should not expect to be able to read from or write to a floppy either.

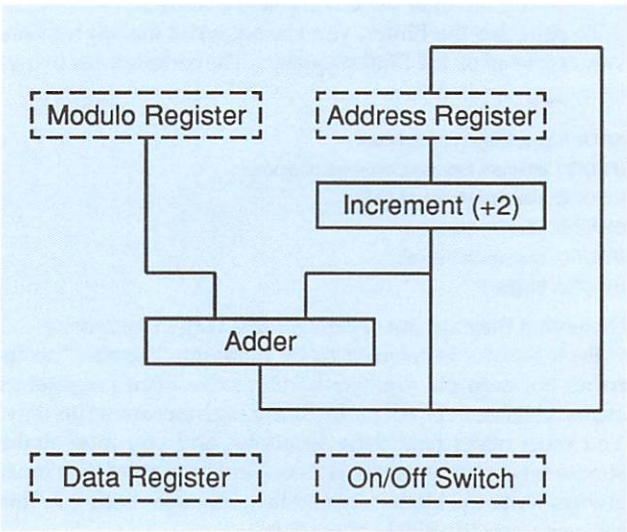


Figure 2: DMA channel details.

Try to use the system-supplied WaitBlit() rather than rolling your own—various versions of the custom chips have problems with the Blitter busy bit and the system software attempts to compensate for these.

THE BLITTER VERSUS INTUITION

The Blitter has no respect for Intuition or the layers system in the Amiga—it operates exclusively on raw memory. Thus, you would almost never use the Blitter to render into an Intuition window, unless you were positive that the window would not be obscured by other windows, requestors, or menus, and that the window would not be sized or moved. Normally this is tough to guarantee.

The Blitter can still be used for rendering to a custom screen. Another common trick is to render to off-screen memory and copy the bits to a displayed window with a function such as BltTemplate().

The graphics routines that use the Blitter—such as BltTemplate()—are smart enough to do their work through layers, just like all of the other graphics functions. But care should be taken that border and gadget imagery not be overwritten.

FOLLOW THE BLOCKS

The key to understanding how the Blitter works is to understand Figure 1 which shows the essential capabilities of the Blitter. We will discuss the individual blocks in this figure in more detail as we go along.

The Blitter has four independent DMA (direct memory access) channels. Three are used to fetch data into the Blitter; the fourth is used to write data back to memory. Typically, each of these channels is responsible for fetching (or writing) a certain rectangular area of memory, letting you combine data from different sources.

Figure 2 gives a more detailed view of one of the DMA channels. Dashed boxes indicate registers that the programmer can write to. Each DMA channel has a 32-bit address register, called bltaptr through bltdptr for channels A through D, respectively. (The current hardware actually uses less than 32 bits, but because the Blitter can only refer to chip memory this does not make any difference.) This address register is interpreted as a byte address—precisely the same type of addressing that the 68000 uses.

After each data fetch, the value in the address register is incremented. Because the Blitter always operates on 16-bit words, the address register is incremented by two. In addition, at the end of each row the value stored in the modulo register is added to the address register. This modulo value is also in bytes, and it is treated as a signed value, so you can use values from –32768 to 32766. Both of these registers must receive even values, as the Blitter cannot access individual bytes from a memory word.

Each DMA channel also includes a switch that you can use to turn that channel on or off. This switch is a bit in one of the Blitter control registers; the bits are BC0F_SRCA through BC0F_SRCC and BC0F_DEST in bltcon0, all of which are defined in hardware/blit.h. If a channel is turned off, no data is actually fetched for that channel. Instead, the channel uses a constant value that you can set by storing into the channel's data register (bltadat through bltcdat).

Now let's consider the bltsize register. It receives the width and height of the blit to be performed. The width is in words and is put into the lower six bits of the bltsize value—a val-

ue of zero actually means 64 words or 1024 bits; the other values are as specified. The height is in bits (rows) and is put into the upper ten bits—here a value of zero means 1024 rows; all other values are as specified. Thus, to perform a blit that is 320 bits wide by 400 rows high, you would store 20 (320/16) as the lower six bits and 400 as the upper ten bits. The 16-bit value can be calculated by multiplying the height by 64 and adding the width in words, so the total value would be 25,620.

As an example, suppose you want to copy a 320x200 one-bitplane low-resolution screen to the upper-left corner of a 640x400 one-bitplane screen, using the A channel to fetch the source and the D channel to write the destination. As shown above, you first call OwnBlitter() and WaitBlit(), then you simply store the starting address of the source screen into custom.bltaptr and the starting address of the destination screen into custom.bltdptr. Setting the appropriate bits in custom.bltcon0 turns on the A and D channels. Because you want to copy the entire source screen, you would set the A channel modulo register to 0.

After each row you copy, you must add 40 bytes (half of 640/8) to move to the next row on the high-resolution screen. So you store 40 into the D channel modulo register. You also initialize custom.bltafwm and custom.bltalwm to hexadecimal FFFF and set the control registers custom.bltcon0 and custom.bltcon1 to 2544 and 0, respectively. (More on how we determined these values later.) Finally, you would write 25,620 to bltsize to start the Blitter. That's all there is to it!

If you wanted to set an entire area of memory to a particular value, you could use the same sort of procedure, only this time you would turn off the A DMA channel (using a value of 496 for bltcon0) and initialize the bltadat to the 16-bit value you wanted to store. You would not need to initialize bltaptr or bltamod, since the channel is turned off.

CONTROL REGISTERS

The Blitter has two control registers—bltcon0 and bltcon1. Each of these registers is 16 bits wide, and any unused bits must be set to zero. So far I have introduced the DMA on/off bits in bltcon0—these are bits 8 through 11 of the register. As I introduce each new Blitter feature, I will indicate what bits of which control register are used.

LOGICAL COMBINATION OF SOURCES

This brings us to the very heart of the Blitter—the combination function that combines data from multiple sources into a single word that can be written to the destination. There are 256 possible functions, some useful, some interesting.

Consider the ways that one bit from each of three sources can be combined to yield an output bit. This function is most easily represented as a truth table. There are eight different combinations of input values; a truth table lists the output bit for each of these combinations. Thus, the function can be described by a single eight-bit value, where each bit corresponds to a particular row of the truth table. Figure 3 illustrates this and gives the details of the combination block in the initial block diagram.

Note that the function described by this truth table is always used, whether or not the DMA channels are on; one bit from each channel—either a bit from a word fetched from memory, or a bit from the preloaded data registers, if the DMA channel is turned off—is used to index the truth table and generate an output bit.

If you prefer standard Boolean logic equations to a truth table approach, you can use them instead, with the following table. Simply look up each product term and combine the values with the Boolean OR for the desired value. The table is used to look up a minterm by finding the section of minterm that references A and B on the top of the table, and selecting the section of the product term that references C (if any) on the side. The values in the table are given in hexadecimal.

	–	A	~A	B	AB	~AB	~B	A~B	~A~B
–		F0	0F	CC	C0	0C	33	30	03
C	AA	A0	0A	88	80	08	22	20	02
~C	55	50	05	44	40	04	11	10	01

For example, to use the logical function AB+~AC, look up AB in the above table and find C0, then look up ~AC and find 0A. If we OR the values together, we get CA hexadecimal, or 202 decimal.

The function AB+~AC arises often. If the A channel fetches a stenciled outline of the object we want to draw with a 0 where we want the stencil to be transparent, B fetches the object itself, and C "prefetches" the background, then we can even draw objects with complex shapes (such as the one in Figure 4) and have the background be visible through any transparent sections. Note that the stencil is necessary to distinguish between white bits in the source that are part of the object and white bits in the source that are meant to be transparent.

This and all other examples in this article consider images only one bit deep. To work with deeper images, whether source or destination, you would simply use one blit for each ►

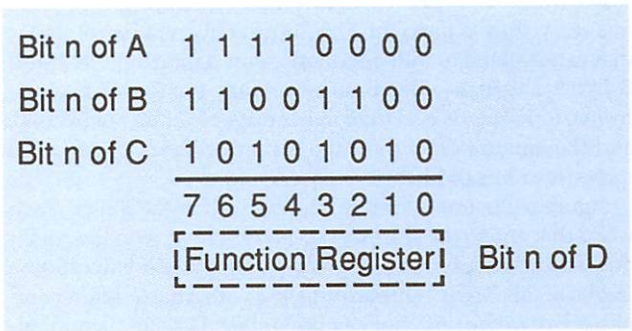


Figure 3: Combination function details.

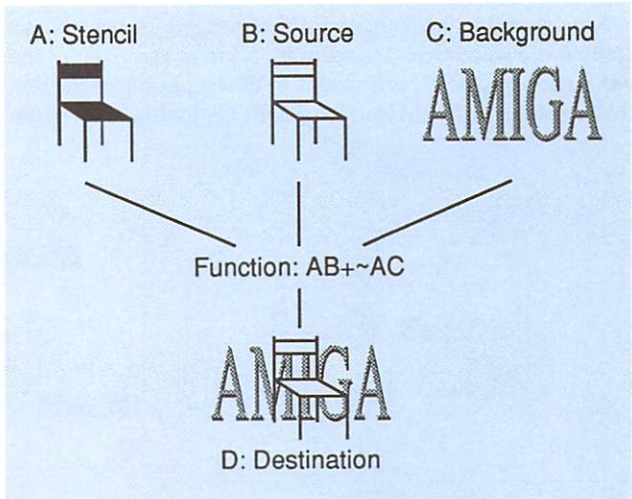


Figure 4: Using "stencils" to draw a complex object.

bitplane. With multiple bitplane images, stencils are especially useful in the way they isolate the transparency information of an object.

The logical function value is stored in the lower eight bits of bltcon0. Our earlier example, where we were just copying the A channel, used a function of just A, or F0 hexadecimal. Combining this with BC0F_DEST and BC0F_SRCA via a logical OR resulted in the required value of 2544. (Using the symbolic names is, of course, recommended.)

If you are starting to get confused, now is a good time to take a break and possibly review our discussion. Things only get more difficult from here.

Most of the difficulty in programming the Blitter stems from the fact that the Blitter operates only on word values and most graphics operations require bit precision. You are responsible for figuring out how to move bit-oriented graphics through operations on words of memory.

The examples so far have all assumed that the images were aligned—that each least significant bit of a source word would be drawn to a least significant bit of a destination word. Usually this will not be the case; you will want to be able to draw a particular image at any bit location on a destination bitmap.

A 16-SPEED SHIFT AND REVERSE

Figure 5 shows an example source image of a small flatbed truck. The source image is six bits high by twelve bits wide, and it is stored in six words of memory. (In this article, a word means 16 bits, or two bytes.) Figure 5 also shows a sample destination bitmap and some possible desired locations for the truck.

Drawing the uppermost truck requires shifting the source image two bits to the right. The Blitter contains a barrel shifter that can be used to independently shift data from the A and B DMA channels—the amount of shift for the A channel, from 0 to 15 bits, is specified in the upper four bits of bltcon0, and the amount of shift for the B channel is specified in the upper four bits of bltcon1.

Figure 6 illustrates the shift mechanism in the Blitter. Each word that enters the shift section is combined with the previous word through that section, yielding a 32-bit value. Some 16 bits of this 32-bit value are selected with a barrel shifter and passed on to the next stage of the Blitter. The shift is normally to the right. Thus, to draw the top truck of the convoy, a shift value of 2 would be used for the source channel.

Note that the "previous word" register does not always contain the word from the memory address right before the word currently being processed. At the beginning of the blit, this register is initialized to zero. At the beginning of each row after the first, the previous word register contains the last word fetched from the previous row—which is not the word before the current one in the current row, if the modulo value for that DMA channel is nonzero.

Now, consider the bottom truck in the convoy. Drawing it requires a left shift of one bit—but the Blitter shifts right! How can we draw it? The answer is to use the descending mode of the Blitter. When you turn on the descending mode of the Blitter by setting the BC1F_DESC bit in bltcon1, the Blitter runs backwards. Addresses are decremented by two rather than incremented by two in each cycle. The modulo values are subtracted not added. Shifts go to the left rather than to the right. This mode will give us the required left shift.

To use the descending mode, the address registers must be initialized to point to the last words of the blit rather than the first words. But, nicely, the modulo values take on the same values they would for an equivalent ascending-mode blit.

The descending mode is also very useful when the source and the destination of a blit overlap—as when scrolling. When scrolling bits up a window, we can use the standard ascending order of the Blitter, because the data from higher memory addresses (which appear towards the bottom of the screen) is written to lower addresses. When scrolling data down a window, however, if we used the ascending mode, at the beginning of the blit we would overwrite a portion of the window that had not been read yet. Rather than a scroll, we would get multiple copies of the first displayed line!

When moving data from one portion of the screen to another such that there is a possibility of overlap, if the source is higher on the screen than the destination, the rows should be processed from bottom to top. If the source is lower on the screen than the destination, the rows should be processed from top to bottom.

In ascending mode, the rows are done from top to bottom and each row is handled from left to right; in descending mode, the rows are done from bottom to top and each row from right to left. You can process the rows from bottom to top, but still do each row left to right— use negative modulos in the DMA channels. In this case, you initialize the address registers to the lower-left memory word of each image, do not set the descending-mode bit, and use a modulo value that is the negative of the sum of twice the width of the bitmap in bytes and the number of bytes between rows (modulo value = –(2*bitmap.width + bytes.between.rows)). In a similar fashion you can use descending mode for each row while using ascending mode row to row—just initialize the address registers to the upper-right word of each image, turn on the descending-mode bit, and use the same modulo value as in the previous example.

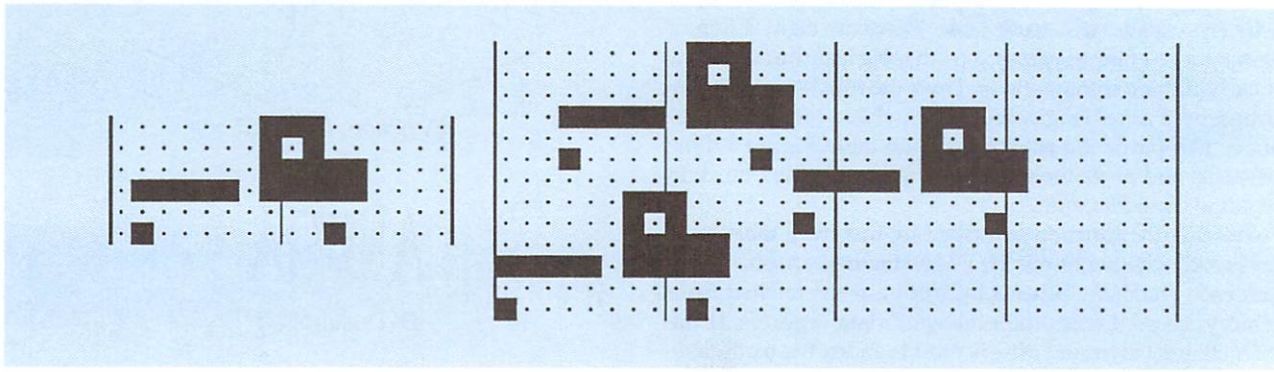


Figure 5: A flatbed truck and a convoy.

Now, consider the lead truck of our convoy. Its image spans two 16-bit words, while the source image only spans one. Thus, though the two images are both exactly 12 bits wide, the source is one word wide and the destination is two words wide (from now on referred to as the "bit-width" and "word-width" of an image, respectively). Similarly, if the lead truck were the source and the smaller bitmap the destination, then you would need to blit two 16-bit words into one 16-bit word! The Blitter only has one width register; what can we do? The solution is to use the largest of all the word-widths. (They will all be within one, because the bit-widths are all identical.)

If we are blitting a source with a word-width of one to a destination with a word-width of two, then we must set the Blitter-width to two words. We use a right shift (and ascending mode). The Blitter will go ahead and fetch data that is from our source image for the second word, however, so we must mask that data out. The Blitter has two registers provided just for this purpose. The bltafwm value is automatically combined via AND with the first word of data from each row fetched from the A DMA channel, and an AND combines the bltalwm with the last word of data from each row. If the row width is just one word, both masks are applied to that word. If you do not want to use any masking, you must initialize both of these registers to –1 or, in hexadecimal, FFFF.

If you use a stencil, you can fetch the stencil with the A DMA channel and set the bltalwm to 0. This way the data fetched from the A DMA channel that is off the stencil (and might contain arbitrary values) will be masked to 0.

RECTANGULAR STENCILS

The Blitter is particularly handy at moving rectangular collections of bits. If we had a stencil it would be a perfect rectangle. Almost always, in this case, you can instead have a "virtual" stencil—created by specific values in bltafwm, bltalwm, and bltadat—with the A DMA channel turned off. Simply set the bltadat register to hexadecimal FFFF for the middle section of the stencil. Next, initialize the bltafwm to the first word of the stencil and the bltalwm to the last word of the stencil, and set the shift register to the same value as that of the B DMA channel, so the stencil is shifted with the data.

In the example at the top of Figure 7, we use a hexadecimal value of FFFF for bltadat (as always), a value of 07FF for bltafwm, and a value of FF80 for bltalwm, and we would shift this channel the same amount as we do the B channel.

If the source has a word-width greater than the destination, you must mask out more than 16 bits on the right (because an extra word will be fetched from the source.) You handle this by orienting the virtual stencil with respect to the destination. Set the A shift value to zero, and set bltafwm and bltalwm to the values they would have if the stencil were shifted to the destination.

Using Figure 7, if we wanted to blit a source with a template given at the top of the figure to a destination located in the bottom, we would use a value of FFFF for bltadat, a value of 0003 for bltafwm, and a value of C000 for bltalwm, with a shift value of 0 for the A DMA channel.

Under certain circumstances, where the source and destination rectangles start at the same vertical location and overlap horizontally (and thus ascending or descending mode is required to not destroy the source before it is read), you cannot set up a virtual rectangle using the A DMA channel in this manner. To resolve the problem, create a single row of the rectangular stencil in chip memory and fetch that with the A DMA channel. Simply set the bltamod value to fetch this row over again for each row of the source.

Space prohibits me from explaining the completely general solution for the problem of copying a graphics rectangle, but the idea is as follows: First, check if a left shift or a right shift is required by comparing the bit position in the word of the starting bit of the source and the destination; set ascending mode if a right shift is required, and set descending mode if a left shift is required. Next, check if the source and destination overlap vertically. (This is only necessary if the source and destination are on the same bitplane.) Decide whether to do the rows top to bottom or bottom to top based on this, and initialize the address and modulo registers appropriately.

The blit-height is simply the height of the rectangle you want to move. Calculate the width required: Add the shift required to the bit position of the leftmost bit of the source to the width, plus 15, and then divide by 16. ((shift.required + (source.x mod 16) + width.in.bits + 15) ÷ 16) Set up a virtual stencil with the A DMA channel, or use a real stencil if you have one, possibly masking out an extra word fetched with the bltalwm register. Choose the function and blit!

PIPELINE REGISTER

For enhanced performance, the Blitter is pipelined. You might expect the Blitter to read all of the sources for the first ►

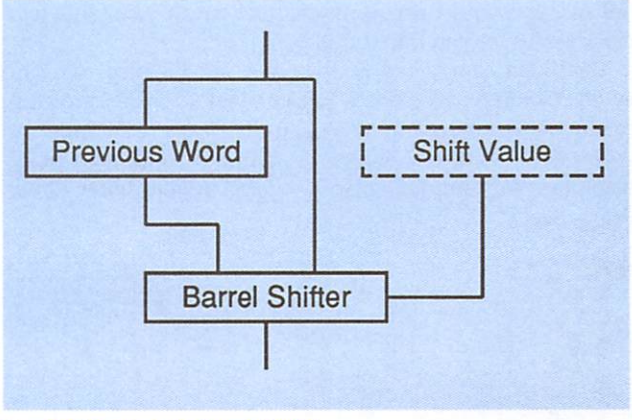


Figure 6: Shift register details (channels A and B).

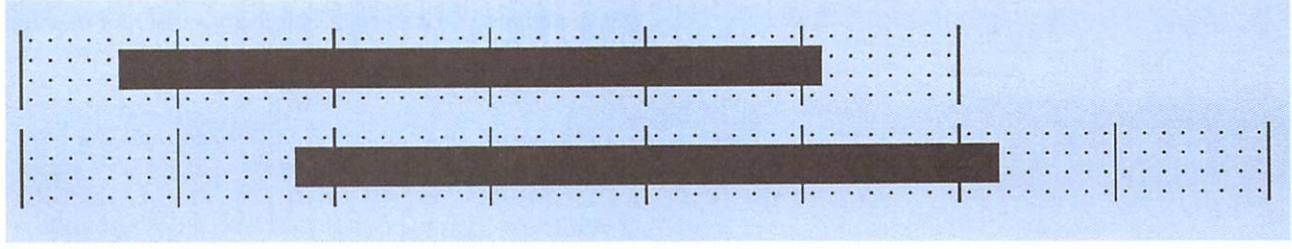


Figure 7: Virtual stencil examples.

cycle, then write the computed value to the destination, then read the sources for the second cycle, then write the computed value to the destination, and so on. Actually, it writes the value computed in the first cycle after reading the sources for the second cycle—no write takes place in the first cycle, and an extra write takes place after the Blitter is finished. This is useful information to know if you are trying to do something like find the bitwise exclusive OR of a large section of memory. For instance, if you wanted to find the bitwise exclusive OR of an array of 1000 16-bit words, you might try setting the A channel to point to the beginning of the array, and the C and D channels to point to the second word, then set the height to 999 and the width to 1 (word), and use a function of A~C+~AC. The thinking would be along the lines of:

```
for (i=0; i<999; i++)
   a[i+1] ^= a[i] ;
return a[999] ;
```

Unfortunately, the pipeline register will cause the "destination" to be written too late. The solution is to set the C and D channels to point to the third word, set the height to 998, and exclusive OR the last two by hand. This will work, and corresponds to the code:

```
for (i=0; i<998; i++)
   a[i+2] ^= a[i] ;
return a[999] ^ a[998] ;
```

PERFORMANCE ISSUES

Choosing which channels to use in a blit can affect the speed of the blit. With all channels turned off, the Blitter requires four 7.18 MHz clock cycles for each Blitter cycle. Using the A DMA channel does not affect the Blitter's speed. Turning on the B DMA channel adds two clock cycles to each Blitter cycle, and turning on both C and D will add another cycle. (Either C or D by itself is "free.") Because almost all blits use channel D, this can be summarized by saying that channel A is free and turning on channel B or C will add two clock cycles to each Blitter cycle.

The Blitter competes with the processor, the display fetches, the Copper, and other DMA channels for access to chip memory—the more bitplanes you display and the taller or wider your screen, the slower the Blitter runs. Using a high-resolution four-bitplane display will slow the Blitter down dramatically.

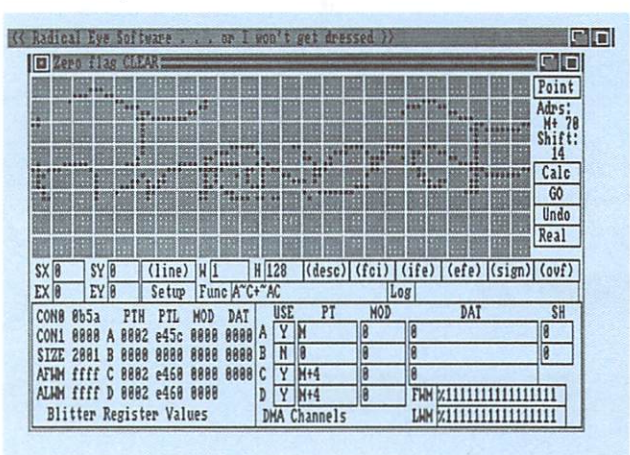


Figure 8: A sample BlitLab screen.

If you are moving large bit images around with the Blitter, it is often much faster to do the middle section that does not need to be masked with a simple A channel to D channel copy, and then finish the two edges with two separate blits that each use the standard procedure of A channel for masking, B channel for source fetch, C channel for destination prefetch, and D channel for writing. This is three blits and a fair amount of overhead rather than just one, but it is much faster for large areas. This trick works whether the source needs to be shifted or not. I'll leave you to work out the details.

To experiment with the Blitter in a safe, controlled environment, give BlitLab (see the Rokicki drawer of the accompanying disk) a try. It provides a gadget for each register or control bit of the Blitter and allows you to set up experiments, perform them, and see the results, without recompiling or crashing the machine. It contains a Blitter simulator as well as access to the Blitter itself, and it comes with a lengthy document on using BlitLab and the Amiga's Blitter, including some information on line mode and area fills. The file blit.tex has details, and Figure 8 shows a sample screen.

WHEN NOT TO USE THE BLITTER

Fast as the Blitter is, it is no substitute for good algorithms. For example, flipping a bit image from left to right can be done with the Blitter—but using a lookup table and the processor is much faster, even on a 68000 system. The Blitter, on the other hand, can flip an image from top to bottom more quickly than the processor.

Careful handling of special cases can also yield faster code than brute-force application of the Blitter. John Conway's zero-player game of life, for example, is a cellular automata simulation that can be done with the Blitter very rapidly. But a more intelligent algorithm that takes advantage of empty areas and periodic elements can dramatically outperform the Blitter—especially when a 68030 is available. For example, exploring the life history of the r pentimino on an Amiga 3000 using the Blitter on a low-resolution, noninterlaced screen takes about 60 seconds. When using the 68030 and a smarter algorithm, this same task requires less than five seconds. On a high-resolution screen the disparity is even greater.

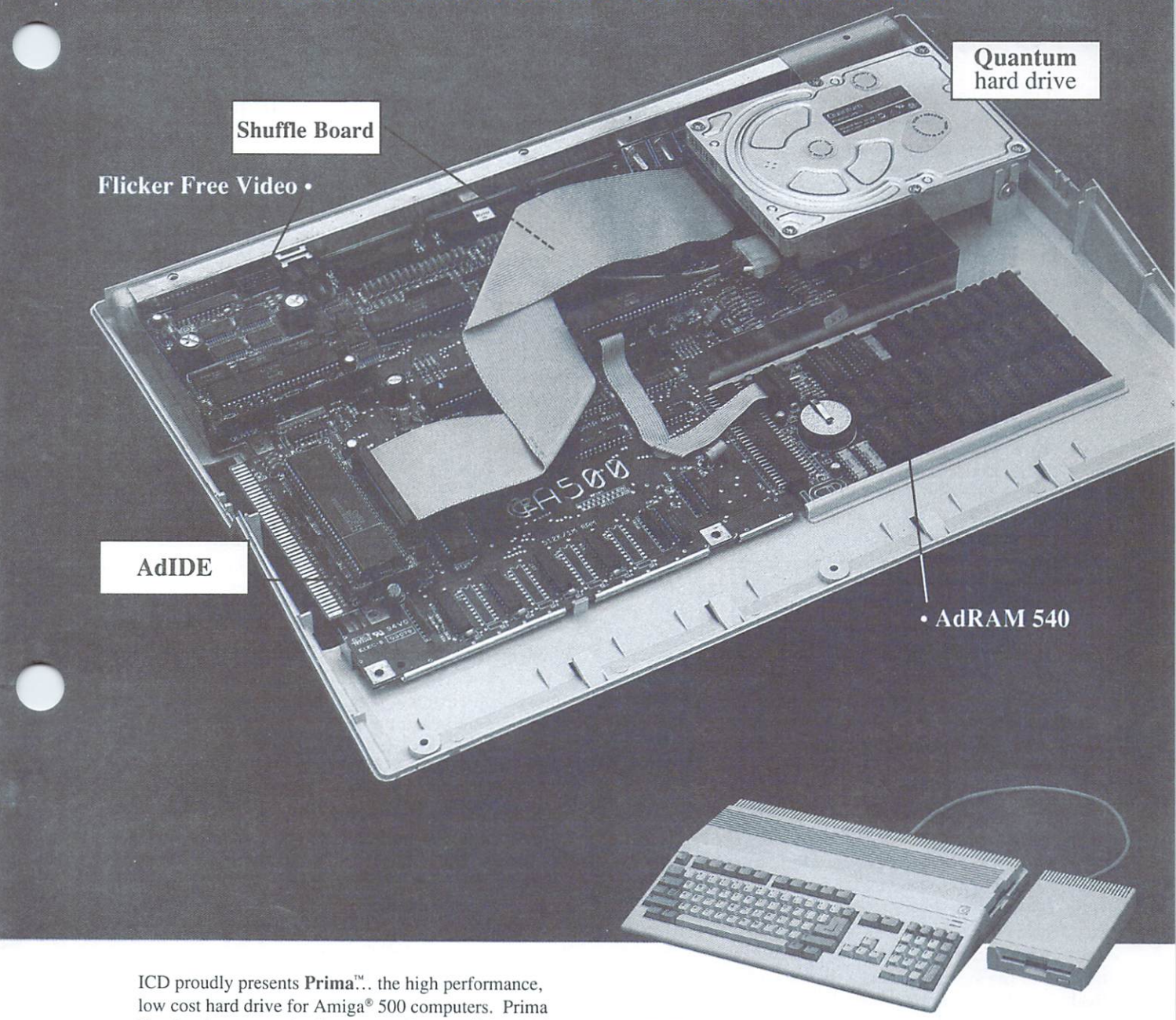
Many problems can be solved more quickly by the Blitter than by the 68000, but on an expanded machine with a 68030 the processor may be faster. You must consider the market for the program you are writing—is it a game that will be predominantly run on A500s, or is it a productivity application whose users will probably have a 68030 processor? If you use the Blitter, the program may not show much of a performance increase when the user upgrades from an A500 to an A3000. For the best overall performance, you could write code that checks the processor type and chooses between processor and Blitter implementations of an algorithm.

In general, using the Blitter is most advantageous on 68000 systems and where its concurrency with the main processor can be exploited. Its built-in barrel shifter, area fill, and masking operations give it an edge over the processor for moving bit images around on the screen. Using the Blitter wisely can really show off the capabilities of the Amiga. ■

Tomas Rokicki, author of AmigaTeX, combines reality and imagination to find radical new solutions to complex problems at Radical Eye Software. Contact him at Box 2081, Stanford, CA 94309, on BIX (radical.eye) or as sysop of the Radical Eye Radio BBS at 415/327-2346.

Prima!

A Look Inside the Ultimate A500.



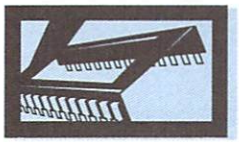
ICD proudly presents **Prima**™.. the high performance, low cost hard drive for Amiga® 500 computers. Prima blends a large capacity, low power Quantum™ hard drive with the **AdIDE**™ host adapter for an unbeatable combination.

Prima replaces the internal floppy drive but includes **Shuffle Board**™ to make your external floppy drive DF0:. **Prima** features auto–booting from FastFileSystem partitions, high speed caching, auto–configuring, and A–MaxII™ support. Formatted capacities of 52 and 105 megabytes are currently available.

Prima comes complete with instructions, software, and all the hardware necessary for a simple, clean, no–solder installation. It does require an A500 with switching power supply, 1 megabyte of RAM, and an external floppy drive for setup and installation.

What other products would we include in the "Ultimate A500"? Of course a four megabyte **AdRAM**™ **540** and **Flicker Free Video**™ with a multi–sync monitor. Why settle for less?

ICD, Incorporated
1220 Rock Street
Rockford, Illinois 61101
USA (815) 968-2228 Phone (800) 373-7700 Orders (815) 968-6888 FAX



DIGGING DEEP IN THE OS

Input on the Amiga

By Michael Sinz

FROM THE DEFINITION—data or information that is read or received by a system—input sounds simple enough. In practice, however, input can be most anything from a data file to the movement of a mouse and many things between. Thankfully, the Amiga contains a rich set of operating system features that make this rather diverse "thing" known as input work in an efficient manner.

The basic input features of AmigaDOS consist of reading to file handles. Thus, these operations can take input ranging from a file on disk to a console window (CON:) to the serial port (SER:). At this level, actions are very far removed and not very operating system dependent, which also means you cannot directly use many of the operating environment's more advanced features. In many cases, simple file I/O precludes direct, interactive action, such as needed by a CAD system, a game, or a word processor.

Because the goal of writing software is to produce an application that solves or helps to solve a problem or perform useful work, often you need to make full use of the operating environment. This is where things get interesting in the Amiga and where much of the power of the system really lies.

In the Amiga OS, a device is a standard interface to a set of code that knows how to deal with a specific set of commands. Various devices control the various parts of the Amiga. For example, audio.device is the interface to the Amiga's audio features. The device interface makes it easy for manufacturers to produce hardware such as internal modem cards, hard drives, and other peripherals that need to connect with the rest of the system. Because the device interface is relatively well defined, applications can easily select between serial.device and modem.device just by changing the string that contains the name.

One of the most interesting devices is input.device, which is at the heart of the Amiga in many ways. Most of the events generated by the user and many that are generated by the system go through input.device's handler chain (also known as the "food chain"). If you can imagine a chain of "filters," much like the components of a stereo system, each of which takes an input signal, modifies it in some way, and then passes it on to the next filter, then you have a basic picture of how the input.device food chain works. Intuition, the Amiga windowing interface system, is basically an overgrown handler on the input.device food chain. The design of input.device and Intuition's interaction with it are what make the event handling system on the Amiga so powerful.

As input events are generated, they get fed down the input food chain. This lets downstream handlers monitor and respond to the events that are passed to them. The handlers may also generate new events that will be passed to any handlers below them. Intuition receives events, then passes them out to the windows that need them in the form of IntuiMessages. While these messages are not input events directly, they are the important parts of the input event.

Input.device is also the main ringleader for the basic input generators, namely the keyboard and the mouse (the keyboard repeat rate, for instance). The "Input Device" and "Keyboard Device" chapters of Commodore's *Amiga ROM Kernel Reference Manual: Libraries & Devices* (published by Addison-Wesley Publishing Company, ISBN 0-201-18187-8) fully explain these other functions of input.device.

HOW TO USE INPUT.DEVICE

While there are many good reasons to use the features of input.device directly, you should first learn when not to use it. For most applications, user interaction should be received by way of Intuition and the message port in the application's window. Usually this is all a program needs, and it makes building an application that works in a consistent manner with the rest of the system much easier. Through Intuition you will get the input events that are directed at your window and the ones that you have asked for only. In other words, Intuition has all the "input-focus" smarts needed in a system where more than one application is running at a time.

Usually, however, does not mean all the time. Some tools, such as the accompanying disk's MiddleButton example (which uses a third mouse button as a type of shift key), need to affect all input events generated. Another example is a tablet or touch-screen driver that makes its device work with all applications by adding the correct mouse or keyboard events to the food chain.

As I stated earlier, Intuition is a handler in the input food chain. It installs itself within the input.device's handler chain at a priority of 50 (more on handler priorities later). Internally, Intuition knows which window is currently active and will pass on input events that are of interest to that window. While I've over-simplified the procedure a bit, that is what Intuition does.

Now, other handlers above Intuition in the food chain can change the stream of input events that flow down to Intuition and thus change its behavior. This, as it turns out, is a very powerful ability. To install a handler above Intuition, you must define a priority of greater than 50 for the handler. The priority field is used to Enqueue() the handler onto the chain. (See the description of the Enqueue() function in the "EXEC" chapter of the *Amiga ROM Kernel Reference Manual: Libraries & Devices* for more details on priority enqueued linked lists.)

"You can build more flexibility in your programs by removing and adding events to the food chain."

Intuition enqueues its handler at priority 50 when the system boots. Because Intuition is the major client of events from the food chain, most handlers would be added above priority 50. Any handler you add at 50 or less will be below Intuition in the chain and thus be unable to affect any activities of Intuition or applications that receive their input from Intuition.

The MiddleButton.asm example program (in the Sinz directory) adds to the system a handler that changes all mouse events to contain a SHIFT qualifier when the middle button on a three-button mouse is held down. This is an example of a handler that is just changing the events that flow down the input food chain. (The MiddleButton program is a useful tool. It works in versions 1.2 and up of the Amiga OS.)

REMOVING AND ADDING EVENTS

Because most of the Amiga's input is received via input.device (and Intuition), the ability to remove or add events provides a simple and system-friendly way to do a number of operations. Two of the Amiga's best known features are its graphics/animation speed and game-playing ability. To make such a program "state-of-the-art," you often need to remove the overhead of parts of the operating system. To make life easier, however, the operating system should be available to help in more mundane or complex tasks, such as loading files and reading the keyboard and mouse. These operations are most efficiently done with the code that knows the machine's hardware best and has been heavily tested.

One way to strike a compromise—remove much of the operating system's unneeded overhead but still let the operating system take care of the other chores—is to grab all the input from the food chain. This involves placing an input handler at a high priority (say, 120) and passing all input events to your application instead of down the rest of the chain. This will prevent Intuition and console.device from getting input from the user and thus causing problems during the critical sections of your code. When you are done with the animation or the game, remove this handler. The system will behave as it did before you installed the handler.

You could use this technique in a high-speed animation player that cannot let Intuition change the view behind its back. The animation program would use Intuition to open the necessary screens and have the user select the actions. When the program was ready to play the animation, however, it would add an input handler to the chain, display the animation without Intuition being able to interfere, and then remove the handler. The handler code could also look for a specific user action signaling that the user wants to stop the animation.

The sample handler in the Sinz drawer, IntuiBlock.c (written in SAS/C) shows how you can get the necessary input events while Intuition is dead. Note, however, that this example contains only part of the instructions needed to hold off Intuition for a while. If there are other applications in the system, you must do more work. Because other applications may call Intuition, you must place yet another lock on Intuition to completely disable it until released. Life gets tricky here; once Intuition is fully dead you must make sure that you do not depend on it in any of your routines. Otherwise, the system will deadlock.

An extreme solution, IntuiBlock.c removes all the input events from the input food chain. You can build more flexibility into your programs by removing only certain events and adding a few of your own to the food chain.

The ability to accept added events gives the system many powerful possibilities. Among the more interesting ones is the ability to change an event of one type to another. For example, MouseKey.c (in Sinz) changes all mouse events it receives to RAWKEY events. MouseKey.c also demonstrates that events added to the chain can be reused when the handler runs again. Because, once given to the food chain, the events belong to the food chain until it is completed, you cannot assume that the events will come back unchanged. Subsequent handlers in the food chain may change any field within the events that they need. Be aware that the example program does little besides install the handler, wait for a CTRL-E signal from AmigaDOS, and then remove the handler. (Developers for CDTV will find MouseKey.c very useful; it eliminates the problem with the mouse pointer movement and the CDTV interface guidelines.)

NEW INPUT DEVICES

MouseKey.c generates input that looks as if it came from the keyboard even though it came from the mouse. Using many of the same ideas, new input devices can inject events into the input.device food chain. These events, if correctly injected, will be indistinguishable from events generated by the Amiga's native input hardware, letting you replace the mouse with a digitizing tablet or a touch screen. This also would allow you to replace the keyboard with, for example, a serial keyboard. Thus, you could build systems for kiosks with special input devices that are, for example, designed for the handicapped or a hazardous environment.

Before OS 2.0, the only way to completely generate these events was for a handler to inject the events into the input food chain. This is not always possible, so you had to do strange tricks to make it work. The only real problem, however, has been the qualifier field. This field can now be loaded from an ►

Become a part of the *AmigaWorld* Programming Team

We're looking for quality programs to support the growth of the *AmigaWorld* product line, and we need your help.

▼

GAMES
ANIMATION
3D
UTILITIES
CLIP ART
AMIGAVISION APPLICATIONS
OTHER STAND-ALONE APPLICATIONS

▲

We offer competitive payment and an opportunity for fame. Send your submissions or contact us for guidelines:

Amiga Product Submissions
Mare-Anne Jarvela
(603) 924-0100
80 Elm Street
Peterborough, NH 03458

outside program by way of the new PeekQualifier() function available in the V36+ input.device. The example program NewEvents.c (in Sinz) generates a few events that are pumped into the input food chain. Note that while the event memory is available after the DoIO(), the entire contents of the event must be reset, because any handler within the input food chain could change the values in the fields.

Many times, input from an external device drives the events that must be sent down the input food chain. Other input devices have software that knows how to talk with the first device and convert its signals to the appropriate events that are then sent down the input food chain. For a simple example of this, check Playback.c in the Sinz drawer.

COMPLEX SYSTEMS

You can put together some very complex systems with the various elements I have described. You may wish to have an external device generate unique input events that an input handler could later use to signal the device to do something or to inject more conventional events into the food chain. Another option is to record the input events and then play them back later. An extremely simplistic form of this can be found on disk in the examples Record.c and Playback.c. With multiple machines, one machine can read the input events and send them over the network to the other machine, thus connecting the input on one to that of the other. This makes it possible to control two (or more) machines with one mouse and keyboard—and food chain.

A NEW STANDARDIZING LAYER

New to OS 2.0, Commodities and Commodities Exchange are a standardizing layer to the operations of the input food chain. Commodities is a library that tries to standardize both the user interface to and the programming model used by tools that work in the input food chain. Commodities also makes it easier for someone who does not understand input.device programming to (at least partially) deal with input.device. This exacts a cost of more CPU usage within the input food chain, however, which affects system performance.

Besides making a simplified programming model and promoting a standard user interface to the tools, Commodities has the Commodities Exchange. This program displays commodities loaded in the system, information on them, and their related hotkey. It also gives the user a centralized interface from which to activate or disable commodities-based tools.

Commodities helps protect the system by making it easier for you to correctly implement global tools (such as 2.0's screen blanker and autopoint commodities). As users continue to mature and more tools such these are built to make the system configureable, commodities may become the dominant method for implementing these. However, there will always be some situations in which the extra insulating layer of commodities would get in the way. As the developer, you must pick the correct method of implementation. ■

Michael Sinz, a Systems Software Engineer at CBM, is responsible for many parts of the OS. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St , Peterborough, NH 03458, or on BIX (msinz).*

ON DISK

Programming HAM-E

By Steve Tibbett

THE HAM-E DEVICE, from Black Belt Systems, gives Amiga programmers two new video modes to complement the many that the standard Amiga chip set already provides. The two extra modes, REGISTER and HAME, loosely compare to the normal lo-res and HAM modes, respectively. REGISTER mode (or REG mode for short) gives you a bitmap of 320×200 pixels with a palette of 256 colors, while HAME mode is easily as confusing as HAM mode was when the Amiga first came out. Both modes can be interlaced, overscanned, and, as you'll see, programmed without much difficulty.

PASS THE COOKIE

The HAM-E device is not mapped into Amiga memory, and has no image memory of its own. It plugs into the Amiga's 23-pin RGB port and patiently monitors the Amiga's normal video output, passing it through unaltered until it sees the "cookie"—a 16-pixel-wide code that the HAM-E interprets as a signal. The cookie can occur anywhere on a scanline, as long as enough room remains on the line for the register information. This signal tells HAM-E whether the frame is to be shown in REG mode or HAME mode. After the device receives the cookie, the rest of that scanline does not make it to the monitor (the user sees black), but the data that the box is blanking out is a combination of pixels that the HAM-E stores internally as 64 colors of the screen's palette. Because HAM-E is not mapped into Amiga memory, the color palette must be encoded into the top scanlines of the image, 64 colors per scanline, with a different cookie indicating that these pixels are to be used as more color information. You may need more than one scanline to transmit the entire color palette—up to four scanlines for 256-color mode.

The color palette that HAM-E uses is eight bits per gun, 24 bits per color register. This means you have a palette of 16,777,126 possible colors—pretty good compared to the 4096 colors maximum that the four-bit-per-gun Amiga chip set gives you. HAM-E can also provide 256 grey levels. With the Amiga's standard 16 levels, the observer can easily see the steps from one level to another, meaning the actual display mode is inserting distracting (and inaccurate) information into the image. With 256 grey levels, the output image contains only visual information that relates to the image, meaning the quality of the image depends on the source, not the source and the output.

HAM-E images are based on eight bits per pixel, a bitplane depth that the Amiga cannot output on its own. The Amiga does have the bandwidth required—a hi-res Amiga screen shows 640 pixels across and can be four bitplanes deep. HAM-E takes two of those hi-res pixels and combines them into one 320-width pixel. The HAME mode also uses the eight-bit pixels to do its magic, so the programming model is strictly 320×200 plus interlace and overscan, with one byte per pixel. Unfortunately, because of the way the image data gets to the HAM-E unit from the Amiga, the data comes out the RGB port with the pixel's eight bits split into four pieces, each piece on one of the four bitplanes. (More on this later.)

In REG mode, you have a simple 320×200 256-color image (or a 320×400 512-color image in Interlace) with no restrictions on color usage or how close together you can put any two colors. The end result from this mode is similar to an IBM's VGA display, but with one important difference. The HAM-E palette is based on 24 bits per pixel, eight bits per gun; VGA is six bits per gun, 18 bits per pixel. Consider it this way: VGA can display 64 distinct shades of red, green, blue, or grey, while the HAM-E can display 256.

You will find that using HAME mode is conceptually similar to using the Amiga's HAM mode (from which HAME got its name). HAME mode takes the eight bits and splits them: The top two bits are the control portion and the last six bits are the data for the pixel. If the control bits are 00, then the data bits pick one of the 60 entries in the palette

Table 1: Graphics.library functions found in the hame.library.

graphics.library	hame.library
SetAPen();	HAME_SetAPen();
SetBPen();	HAME_SetBPen();
ReadPixel();	HAME_ReadPixel(); (to get Pen number)
	HAME_GetRGBPixel(); (to get 24-bit RGB value)
WritePixel();	HAME_WritePixel();
SetRGB4()	HAME_SetRGB8();
DrawEllipse()	HAME_Ellipse();
Draw();	HAME_Line();
RectFill();	HAME_Box();
WritePixelArray8(); (KS 2.0)	HAME_8BitRender();
ScrollRaster();	HAME_Scroll();
Text();	HAME_Text();

"Black Belt provides two libraries of functions—hame.library and renderhame.library."

and use it directly. If the control bits are 01, 10, or 11, then the data bits replace the Red, Green, or Blue component of the previous pixel with the data section of this pixel.

As I said in the REG mode discussion, you can load up to 256 color registers in the HAM-E hardware itself. In HAME mode, however, you cannot get at all of these with only six bits to choose a register. To let you access them, the values 60, 61, 62, and 63 are not interpreted as "get this register's RGB values." Instead, they cause the hardware to change to a new "bank" of 64 registers of which you can access 60 (0–59) out of the 256 available. The colors for a pixel that contains any one of these four special values are held in the R, G, and B values from the previous pixel. This capability provides for up to 240 preset 24-bit registers (or 480 in interlace) to be used to "fix up" a HAME mode image, if you follow some simple rules regarding how you display the image. For example, the screen can be divided into four vertical "quadrants," and a different group of 60 registers be used to fix up each quadrant. The result will be a much more accurate image as far as HAM artifacts are concerned, and no difficult issues about where and how to switch banks.

If you weren't watching your math carefully you might say it's possible to have all 16 million colors on-screen at once—not likely, as you have only 128,000 pixels in a 320x400 resolution! The number of colors on-screen, however, is limited only by your image and how well it works within HAME mode's limitations. You can do some very impressive things with this many colors!

The HAM-E device supports Interlace, and because the Amiga's interlace mode is simply two normal images shown half a scanline apart, you can load the device with an entirely different color palette on the even and odd frames of an interlace picture. This translates to 512 colors for a 320×400 screen in REG mode and also results in a better HAME image, because the base palette for each of the interlace fields can be different.

NEW BUT FAMILIAR LIBRARIES

While you should understand how HAME-mode images are built, most of you will never need to manipulate the bits that make up one directly or create the cookie and the color palette data. Rather than force each Amiga programmer to create HAM-E rendering routines and color-palette routines in HAM-E format, Black Belt provides two libraries of functions. Using these libraries, you can write code for HAM-E without ever dealing with the data as anything except 24-bit data or as eight-bit data with a 24-bit 256-color palette.

The first, hame.library, contains routines that make programming HAM-E feel like programming the native Amiga modes. If you have programmed with the standard graphics.library routines, the hame.library routines will look familiar. (For a complete list of equivalents, see Table 1.) When using hame.library, however, you pass a pointer to a HamePort instead of rendering to a RastPort. You create a HamePort by calling the HAME_Init() function and passing to it both a pointer to an open hi-res screen and information about how you want the screen to look (the number of cookie lines, the size of a Blitter buffer to allocate for the routines to use, and the mode for the screen).

Because the HAM-E display is generated by the standard Amiga output, and because a completely blank scanline will cause the device to stop interpreting the display data as a HAM-E mode and return to passing it through, all the HAM-E modes work very well on a standard Amiga screen. You can drag, depth-arrange, open, and close HAM-E screens as you can any standard Amiga screen. You must keep one thing in mind, however, when opening a HAM-E screen. You have to add the number of cookie lines the screen will need to the screen's height. The number—from one to four—depends on the mode and is automatically taken into account by all the drawing routines in hame.library. For example, if you want to render a REG-mode image with 256 colors to a screen that is 320×400, you must open a 640×408 Amiga screen with ViewModes of (HI-RES | LACE) and pass this to HAME_Init(), which converts your screen into the proper HAM-E mode screen by adding the cookies. When you draw to the screen through the library, the number of cookie lines is automatically added to any Y value you pass, so you cannot accidentally overwrite the cookie.

The library routines for handling the palette are particularly important because of the way the HAM-E palette is kept as part of the image. For example, consider color cycling. Calling HAME_SetRGB8() for 256 colors is not hard, but the results are too slow if you are cycling once every few vertical blanks, as color cycling normally demands. The library provides a much better method: routines that cycle a range of the palette to the left or right by one color. The routines are HAME_CycleLeft() and HAME_CycleRight(), respectively.

Hame.library handles standard Amiga ColorFonts and has routines for opening them (HAME_OpenFont()), getting the palette from the font (HAME_GetFontPallete()), and mapping the colors the font requires into the colors that your current palette has available (HAME_SetFontPen()). A call to HAME_QText (after a setup call to HAME_MakeFTables()) greatly speeds up text rendering, if you don't mind a few restrictions: You can render only in color numbers that are mul- ►

tiples of 16 and in a font of 16 colors or less. (All the library routines also work with nonColorFonts.)

Another group of functions in the library is for manipulating "clips"—rectangular chunks of image data that are conceptually similar to the Image structures that some of the intuition.library functions take. The library includes HAME_GetClip() and HAME_PutClip() for grabbing a clip from a bitmap and putting it back somewhere else; HAME_ReadClipPixel and HAME_WriteClipPixel for peeking into or changing a clip's image; and similar routines for dealing with a clip's "mask." A mask controls where the clip is transparent. Obviously, there are also routines for building the mask and disposing of it when you are finished.

TWO TYPES OF PROGRAMS

Programs for the HAM-E fall into one of two categories, based on the type of screen updates required. For programs that create an image to be shown once, speed is not important but image quality is vital. For real-time programs that need to update portions of the screen or even the whole screen often, speed is vital. The first type, the image generators, can be compared with the raytracers and the Mandelbrot-set generators we currently have. They generate an image and present it to the user, either line-by-line or all at once, when the image is completely generated internally. The second type is everything else—programs that use the HAM-E modes as their main mode of operation (including user interface) for animation or other dynamic displays.

Image generators can make good use of HAME mode's extra colors and are not affected by the mode's drawbacks, because nothing is happening on top of the images. The second HAM-E library, the renderhame.library, was designed for these programs. This library has only three functions: One to render images into HAME mode, one to render to REG mode, and one to save the result to an IFF file (provided as a convenience, because any normal ILBM screen-save routine will do). The most useful routine here is HAME_RenderHam(), which creates an image out of 24-bit raw data in HAME mode—a great deal of work. In a similar fashion, HAME_RenderReg() converts 24-bit data into REG mode. Because you do not supply the HAME_Render functions with a palette, they must create one. While you can call these render functions one scanline at a time, to get the best results you must call the library again with the entire image. The function can pick a better palette after examining the entire image, instead of just one scanline. Therefore, you are better off waiting until the image is finished and calling the render function only once.

Dynamic programs that will want to change an image after it's placed on the screen (paint programs, animation programs, graphic databases, or programs with buttons, magnifiers, or similar user-interface elements) are best off using the 256-color mode. In this mode, all the rendering is very straightforward. Simply blitting a graphic into place or drawing using the provided library routines are all that are needed to keep the display as you intended and free of unexpected artifacts.

If you restrict your extra rendering in HAME mode to the first 60 registers in the palette (so you do not use the special palette-changing colors or set either of the top two bits, causing a Hold-and-Modify effect) and if you write another known color a pixel to the right, then you know the user will be able to read or recognize what you are doing. The colors to the right of the rendered area may be distorted, but for operations that will not be permanent (such as dragging out a box) this may be fine. If it is not, you have only a few choices. You can render in a window of a solid color on the HAME mode screen. (Making the window touch the right side of the screen will keep colors from being distorted, but this restriction is a major limitation.) Saving the background behind the image will cause no permanent damage. You can take the route that Black Belt did with their Paint and IP programs—placing the a smaller screen in front of the main screen, leaving most of the main screen visible but still providing a panel of controls. Finally, you can keep the data that generated the screen around, so you can rerender portions of the screen on demand.

Choosing which video mode to use for your application and whether to make your controls available on the HAM-E screen itself or on a separate panel are very important decisions. They will affect every routine that deals with your program's user interface. Plan carefully in advance; weigh the many advantages of REG mode against the extra color depth of HAME mode. This quandary is not new to HAM-E; standard HAM mode is rarely used for any work in which the image is animated (under program control) or changing, while the normal lo- and hi-res modes do not provide enough colors for images created by raytracers and such.

PLUS EXTRA

The HAM-E Plus has an extra switch on the front for activating what Black Belt calls the Anti-Alias Engine, or AAE. The AAE simply inserts another pixel between every two pixels, doubling the horizontal resolution of your image. Unfortunately, you do not have direct control over what this pixel is, and it only works in the horizontal direction. A plus is that the AAE is completely transparent; the switch is external, and the user can flip it on or off to see what it does for a given image. You do not have to change older images to take advantage of it. The only minus is that the switch is external and is not under software control.

SLOW DRAWBACKS

There are a few things to keep in mind when writing HAM-E code, all of which relate to speed. The biggest problem with doing any real-time work on HAM-E is that it uses a 640x200 four-bitplane hi-res Amiga screen to build its display. The Amiga is set up to share chip memory with the processor. If you go beyond a lo-res screen with four bitplanes or a hi-res screen with two bitplanes, the DMA required to update the display starts cutting into the time the processor is allowed on the chip RAM bus. With a four-bitplane hi-res display, such as the one the HAM-E requires to build its display, the CPU is not allowed to touch the chip RAM bus during bitplane DMA. Anytime you touch chip memory to update the display, the processor will have to wait until either the horizontal-blanking or the vertical-blanking period. As an example of the impact, Jez San's SPEED V2.0 clocked an Amiga 3000 with a HAME image displayed and the test code running in fast RAM at 6.75 times faster than an Amiga 1000. The same benchmark resulted in a rating of 0.96 when the code ran in chip RAM—at about a seventh of its former speed, the A3000 was actually slower than the A1000.

With this in mind, optimize your rendering as much as possible, so the minimum number of pixels are actually drawn to the screen. If you write code for the 68020 and above proces-

sors only, then you can depend on fast RAM being much faster than chip RAM and, therefore, should render your image entirely into fast RAM. This will require your own set of drawing routines, however, as both the graphics.library and hame.library use the Blitter for graphics operations and the 68030 can easily outdo the Blitter for moving graphics around—especially if the graphics are in fast RAM.

Another drawback to the HAM-E display is the mouse pointer, because of the way the graphic image is actually transmitted to the HAM-E device. The normal Amiga mouse pointer is a sprite that floats freely over the bitmap below it. The sprite hardware fetches the sprite image from memory and shows it instead of the contents of the bitmap underneath it. The sprite is always independent of the display. Because the HAM-E modes depend on the actual output from the custom chips rather than the image in RAM, the HAM-E has to think that the sprite is part of the image itself. This is acceptable except in two situations: HAME mode and the palette.

Because the image in HAME mode is made up partially by changes in color values from the previous pixels, having a mouse pointer in the way will "fool" the device into thinking the pointer is the previous pixels instead of the pixels to the right of the mouse pointer. As a result, in a HAME-mode image you may get little streaks extending to the right from the right edge of the mouse cursor, depending entirely on the image. You can easily prevent this effect by turning off the mouse cursor when your program notices that the cursor is overlapping the HAME display. Starting another process to constantly check the mouse cursor location and blanking it when appropriate would totally eliminate this problem without affecting the way the body of your program operates.

The other thing the mouse cursor can do, with unsightly results, is obscure portions of the cookie lines or even the cookies themselves. Obscuring the palette portion of the lines will change colors in your image—parts of the image will flicker into different colors as the sprite hides the palette data behind it. If the sprite obstructs all the cookies, then the entire image will cease being a HAM-E image and will be shown as a rather wrong-looking hi-res image!

You have your choice of simple preventive measures. The best is to use the same bit of code you use to blank the cursor when it affects the image. An easier way, although it still allows the user to change individual palette entries, is to limit the height of the pointer sprite to one less than the number of cookie lines that you have. This won't work for HAME-mode images where you typically have only one cookie, but for REG mode 256-color images, it will ensure that there is always at least one scanline worth of untouched cookie to keep the display from switching out of a HAM-E mode.

For working examples of the techniques I've discussed, check out the Tibbett drawer on the accompanying disk. Example1 generates a 320x200 24-bit bitmap and renders it to both HAME-mode and REG-mode displays. A dynamic program, Example2 shows a simple user interface for a 256-color REG mode display. ■

Steve Tibbett is the programmer of VirusX and sysop of the AmigaZone on Portal (SteveX). During the day, he writes games for Artech Digital Entertainments. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (s.tibbett) or Portal.*

Extending ARexx

With the help of a few libraries, you can build menus and requesters for your ARexx programs.

By Marvin Weinstein

AREXX, WHILE ADVERTISED as the Amiga's universal scripting language, is actually a full programming language. Thanks to its simple syntax and powerful string manipulation capabilities, it is well suited to writing applications that interact with the user and is an ideal tool for customizing your working environment. The only drawback to using ARexx for these purposes is its lack of built-in functions for accessing the Amiga's graphical user interface (GUI). Fortunately ARexx was designed to be easily extendable and offers several ways to get around this limitation—ARexx host applications, ARexx function hosts, and ARexx shared libraries.

While I will mention all three methods, we will focus on function hosts and shared libraries. To demonstrate accessing a function host or shared library, I included the executable for a simple function host (quicksort in the usrc subdirectory) and two freely distributable ARexx shared libraries (rexxarplib.library, rexxmathlib.library in the libs subdirectory) in the Weinstein drawer of the accompanying disk. This article is meant to be read in conjunction with the examples on the disk; they are extensively commented.

HOST APPLICATIONS

If you normally think of ARexx as a scripting language for host applications, it might seem the "tail is wagging the dog" when I suggest considering a host application as an extension to the ARexx language. Thinking of ARexx as just a universal scripting language, however, tends to focus too narrowly on automating procedures for a single application. This overlooks the benefits obtained by having independent applications work together. On the other hand thinking of a host program as an extension of ARexx suggests combining multiple applications into a custom package, as they are called from a single ARexx program. The debate, however, is a topic for a different article. At this time, we need only review how to send commands to a host application and contrast this with the process used for function hosts and shared libraries.

A host application is one that has opened a public message port to which ARexx messages can be sent. Such messages instruct the application to perform a specific action. To send a message to a host application from within an ARexx program you use the address instruction:

```
address HOSTAPPLICATIONPORT "message"
```

Upon receipt of a message, the host application processes it and replies to the sending program indicating success or failure. It is up to the host application to decide how to handle commands that it does not understand. The simplest option for the host application is to do nothing but return an error code. The basics of identifying and communicating with host applications were discussed in my earlier article "ARexx Arcana: Hosts and Quotes" (p. 2, August/September '91).

FUNCTION HOSTS AND LIBRARIES

You do not use the address() instruction to communicate with a function host or shared library. If an ARexx program calls a function whose name does not match the name of an internal subroutine, then ARexx forwards this call, in turn, to each of the entries on the *library list* it maintains for this purpose. The process continues until ARexx finds a host or library that understands the function call or until the list is exhausted. If ARexx does not find a match, it looks for an ARexx program with the same name. If one is found it is executed. Otherwise ARexx returns with the message "function not found."

Function hosts and libraries are appended to or removed from the library list using the function calls:

```
addlib(portname, priority)
addlib(libraryname, priority,offset,version)
remlib(libraryname)
```

The first addlib() call adds function hosts, and the second adds shared libraries. The difference exists because a function host has a public message port to which ARexx forwards function calls and a library does not. When you add a function host to the library list, you simply add the name of its public port and assign it a priority. Programs can access functions stored in the library by knowing the location of the library's *dispatch function*. The offset argument to the addlib() function provides this information to ARexx. It is an integer that is, in principle, library dependent and that should be supplied to you along with the ARexx shared library. For all of the libraries provided with this article it is a *negative* integer, –30.

Because libraries get updated from time to time they have embedded version numbers to identify them. By specifying a value for the argument version in the call to addlib(), you tell ARexx not to add any library to the library list if its version number is less than the value specified for version. Generally, specifying 0 is perfectly fine.

Be careful: The addlib() function merely adds the name of the function host (or shared library) to the library list. *It does not check to see that the port or library actually exists.* Thus, a return code of 1 from the addlib() call does not mean that everything is all right. Programs can still break when ARexx attempts to access the function host or library. This has been a source of much confusion to ARexx novices who call addlib() ►

FOR ORDERS AND INFORMATION IN USA & CANADA CALL **1-800-759-6565**

Order Hours: Mon-Thurs, 9am-7pm/Fri, 9am-5pm/CLOSED Sat/OPEN Sun, 9:30-6(ET)

AMIGA 500 EXPANSION KIT $119

SUPRA RAM 500 512K RAM EXPANSION with CLOCK, CALENDAR and MASTER 3A-1 EXTERNAL DRIVE

MONTGOMERY GRANT™

AMIGA WORLD TECH JOURNAL

OUTSIDE USA & CANADA CALL **(718)692-0790**

OR WRITE TO: MONTGOMERY GRANT: MAIL ORDER DEPT. 33 34th ST., DEPT. A, BROOKLYN, N.Y. 11232 FAX #7186923372 / TELEX 422132 MGRANT

ESTABLISHED 1967

RETAIL OUTLET PENN STATION, MAIN CONCOURSE (Beneath Madison Sq. Garden) NYC, 10001

Store Hrs: MON-THURS, 9:30-7 / FRI, 8:30-5 / SAT-CLOSED / OPEN SUN, 9:30-7

VIDEO TOASTER

CALL FOR OUR LOW, LOW PRICE

NEWTEK VIDEO TOASTER BOX ALL IN ONE VIDEO PRODUCTION SYSTEM **CALL**

PERSONAL TBC.....$729

MANY OTHER TIME BASE CORRECTORS (TBC) AVAILABLE

TOASTER TUTORIAL VIDEO CASSETTE $19.95

AMIGA™ 2000

1MB EXPANDABLE TO 9MB DISK DRIVE BUILT-IN 3.5" SYSTEM SOFTWARE MOUSE

$1199

AMIGA 2000HD....$1499 AMIGA 2500/50....$2749

AMIGA 2500/100.....$3229

AMIGA™ 3000

ALL MODELS AVAILABLE Starting as low as **$2299**

AMIGA™ 500 $469

AMIGA 500P.....CALL

AMIGA PERIPHERALS

A-2088D XT BRIDGEBOARD.....$449
A-2286D AT BRIIDGEBOARD.....$599
A-2630/4 ACCELERATOR KIT (25MHz, 68030, 4MB, 68882).....$1499
A-1680 MODEM w/CABLE.....$69

AMIGA 500 & AMIGA 2000 COMPATIBLE HARD DRIVE PKGS.

MIX 'n MATCH THESE SCSI CONTROLLERS & HARD DRIVES TO FIT THE RIGHT PACKAGE FOR YOU!

A-500 SCSI CONTROLLERS

DATAFLYER 500.....$125
RAPID ACCESS TURBO A-500.....$229
TRUMPCARD 500.....$159
TRUMPCARD 500 PRO.....$229
XETEC FASTTRACK A-500.....$219

A-1000 SCSI CONTROLLERS

XETEC FASTTRACK A-1000.....$299

A-2000 SCSI CONTROLLERS

ADSCSI 2080.....$159
CA MALIBU BOARD 2000.....$109
DATAFLYER 2000.....$85
GVP SERIES II HC A-2000.....$149
GVP SERIES II HC8/0 A-2000.....$195
RAPID ACCESS TURBO A-2000.....$229
SUPRA WORDSYNC.....$99
TRUMPCARD 2000.....$99
TRUMPCARD 2000 PRO.....$135

HARD DRIVES

SEAGATE ST-157N-1 (49MB).....$215
SEAGATE ST-1096N (80MB, 3.5").....$325
QUANTUM 52MB (LOW PROFILE).....$235
QUANTUM 105MB (LOW PROFILE).....$379
QUANTUM 170MB.....$619
QUANTUM 210MB.....$699

GVP

GVP RAM 8/2 A-2000 (2MB EXP. to 8MB).....$175

NEW SERIES II

A-2000 22MHz/1MB exp. to 13MB/68882...CALL FOR
A-2000 33MHz/4MB exp. to 16MB/68882..LOW PRICE

CALL FOR LOW, LOW PRICES

SCSI HARD DRIVE PACKAGES AVAILABLE

GVP 3050 Kit (50 MHz) w/68030, 4MB exp. to 32MB, 68882.....$2159
GVP 3050 KIT w/QUANTUM 40MB.....ADD $300
GVP 3050 KIT w/QUANTUM 80MB.....ADD $500
GVP 3050 KIT w/MAXTOR 210MB.....ADD $950
GVP A-500 HD 8+0/52Q.....$554
A-500 HD 8+0/105Q.....$804
RICOH 50MB Removable w/Cartridge.....$759

AMIGA COMPATIBLE PERIPHERALS, ACCESSORIES & SOFTWARE

A-MAX EMULATOR II.....$129
AMIGA 3000 32 Bit Memory.....AVAILABLE
AMIGAVISION SOFTWARE.....$89
AMIGA 1.3 ROM (8850).....$49
AMIGA 1MB FATTER AGNUS CHIP(8372A).....$99
APPLIED ENGINEERING
1.52MB HI-DENSITY DRIVE.....$199

ATonce PC/AT EMULATOR.....$249
A-2000 ADAPTOR.....IN STOCK

BASEBOARD

2MB Daughter Board Available

Memory Expansion for A-500 (uses A-501 Exp. Slot)

0K.....$99 1MB.....$135 3MB.....$229
512K.....$119 2MB.....$175 4MB.....$259

BOMAC TOWER A-2000.....$239
BROADCAST TITLER II.....$216

BODEGA BAY

By CALIFORNIA ACCESS

EXPANSION CONSOLE Turn your A-500 into a A-2000 Compatible *NOW AVAILABLE* **CALL**

CHINON INTERNAL DRIVE for A-2000.....$79
COLOR SPLITTER.....$105
DAKOTA SKETCHMASTER 12x12.....$369
DAKOTA SKETCHMASTER 12x18.....$569
DIGITAL CREATIONS DCTV.....$369
DIGIVIEW GOLD v.4.0.....$114
EXCELLENCE 2.0.....$105
EXPANSION SYSTEMS
Heavy Duty Power Supply for A-500.....$79
FLICKER FIXER.....$229
FLICKER FIXER PAL.....$269
FLICKER FIXER GENLOCK OPTION.....$39
FLICKER FIXER DEB2000.....$89
FRAMEGRABBER.....$419
FRAMEGRABBER 256.....$499

GENLOCKS

MINIGEN.....$184
SUPERGEN.....$599
SUPERGEN 2000S.....$1339
VIDTECH SCANLOCK.....$759
VIDTECH VIDEOMASTER.....$1049

GOLDEN IMAGE

HANDSCANNER w/MIGRAPH TOUCH-UP.....$239
OPTO-MECH MOUSE.....$35
OPTICAL MOUSE.....$49

HAM E.....$295
HAM E/PLUS.....$379

ICD

ICD ADIDE.....$89
ICD **NOVIA 20I** 20MB INTERNAL DRIVE for A-500.....$479
ICD ADSPEED Excellerator (14.3MHz.).....$205
ICD FLICKER FREE VIDEO.....$279
NEW PRIMA 52I.....$429
NEW PRIMA 105I.....$599

INSIDER II Internal Memory for A-1000

OK Expandable to 1.5MB

512K.....$219 1MB.....$249 1.5MB.....$279

IMAGINE.....$189
LATTICE C5.1.....$189
LATTICE C++.....$99
LIVE! A-2000.....$279
MASTER 3A-1 3.5" DISK DRIVE.....$79
MEGA-MIDGET ECONOMY (25MHz).....$489
MEGA-MIDGET ECONOMY (33MHz).....$559
MEGA-MIDGET RACER (25 MHz.).....$599
MEGA-MIDGET RACER (33 MHz.).....$679
MEGA-MIDGET RACER (50MHz).....$849
PAGESTREAM 2.1.....$164
PANASONIC 1410 CAMERA.....$179
PROFESSIONAL PAGE 2.0.....$215
PRO VIDEO GOLD.....$129
PRO VIDEO POST.....$139
PROWRITE 3.1.....$89
SCALA.....$257
SCULPT ANIMATE 4D.....$289
SHARP JX 100 COLOR SCANNER w/SOFTWARE & CABLES.....$645
TURBO SILVER 3.0.....$58
WORDPERFECT (AMIGA).....$155
XETEC CD ROM CDX-650E EXTERNAL.....$649

MONITORS

NEC MULTISYNC IIID.....$589
NEC MULTISYNC IIIDS.....$609
SONY 1304 MULTISYNC.....$649
SEIKO 1440.....$509
SEIKO 1445.....$579
SEIKO 1450.....$629

VIDEO PACKAGE

• PANASONIC PV-1410 VIDEO CAMERA COMPLETE w 16mm LENS • COPYSTAND w LIGHTS • DIGIVIEW GOLD 4.0

w/ FIXED LENS.....$309 w/ VARIABLE LENS...$339

Supra 500XP HARD DRIVE KITS

512K, 20MB.....$349 512K, 80MB.....$559
512K, 52MB.....$475 512K, 105MB.....$655
2MB, 20MB.....$459 2MB, 80MB.....$639
2MB, 52MB.....$499 2MB, 105MB.....$715
2MB, 52MB (1MBx4).....$555 2MB, 105MB (1MBx4).....$725

2MB THRU 8MB VERSIONS AVAILABLE

SUPRA RAM 500RX 1MB $128 2MB $188

512K EXPANDABLE TO 8MB-PASS THRU BUS

SUPRA 2400 EXTERNAL MODEM w/CABLE.....$99
SUPRA 2400 ZI.....$159
INTERNAL MODEM.$114
SUPRA 2400 MNP....$145
SUPRA 2400 PLUS w/MNP5, V.42, bis.$165
SUPRA 2400 ZI PLUS.....$159
SUPRA 9600 PLUS.....$549

SUPRA 3.5" EXTERNAL DRIVE.....$98

SUPRA RAM 2000

OK.....$105 4MB.....$235 8MB.....$369
2MB.....$169 6MB.....$299

SUPRA RAM 500 512K Expansion for A-500 **$47**

NEW COMMODORE CD-TV CALL

PRINTERS

PANASONIC

KXP-1180.....$159.95
KXP-1123.....$234.95
KXP-1124I.....$299.95
KXP-1624.....$369.95
KXP-1654.....$589.95

STAR

NX-1001.....$149.95
NX-1020 R.....$189.95
NX-2420.....$279.00
NX-2420 R.....$299.00

CITIZEN

GSX-140.....$275.95
GSX-145 Wide Carraige.....$385.95
200GX.....$159.95
Color Option Kits.....CALL

CANON

BUBBLEJET BJ300...$499 BUBBLEJET BJ330...$679

HEWLETT PACKARD

DESKJET 500.....$499
LASERJET IIP w/Toner.....$899
LASERJET IIIP w/Toner.....$1099

COMMODORE MPS 1270 INK-JET.....$139

FOR CUSTOMER SERVICE or ORDER STATUS CALL: (718) 692-1148

CUSTOMER SERVICE HOURS: MON-THURS 9am-5pm/FRI 9am-4pm/SUN 10am-4pm

NO SURCHARGE FOR CREDIT CARD ORDERS

WE INVITE CORPORATE AND EDUCATIONAL CUSTOMERS. DISCOUNTS FOR QUANTITY ORDERS. RUSH, 2nd DAY & NEXT DAY AIR SERVICE AVAILABLE.

CUSTOMER TOLL FREE TECHNICAL SUPPORT

Certified check, Bank Check Money Orders, Approved P.O.s, Visa, Mastercard, Amex, Optima, Diners Club, Carte-Blanche, COD's & Wire Transfers accepted. Please call before submitting P.O.s. Non-certified checks must wait 2-4 weeks for clearance. Prices and availability subject to change without notice. Not responsible for typographic errors. Return of defective merchandise must have prior return authorization number or returns will not be accepted. Shipping & Handling additional. Second Day & Next Day Air available at extra cost. Canadian orders please call for shipping rates. APO/FPO orders please add 10% shipping & handling - min. $15 (Over $1200-8%, Over $3000-6%). All APO/FPO orders are shipped first class priority air. Oversized shipments may be levied a surcharge. All orders can be shipped air express, Call for details. We check for credit card theft. DCA #800233. Amiga is a registered trademark of Commodore Amiga, Inc.

for libraries that they have forgotten to install in their libs: directory. To see what entries are on the library list use the rexxsupport.library function show() by typing:

```
rx "say show('L');"
```

in a CLI. (You can leave out the rx, if you're running WShell).

The order in which ARexx traverses the entries on the library list is determined by the priority assigned to each entry and by the order in which the items were added to the list. Items with higher priority come ahead of those with lower priority; items with the same priority are arranged in the order they were added to the list.

WARNINGS AND WORK

Do not assume that, because a function host is a program that has a message port, you can write function hosts in ARexx. Unfortunately, this is not the case. A function host must be able to notify the REXX process if it does not know about a function, so that it can continue down the library list looking for someone who does. While the rexxsupport.library gives you some functions for manipulating ARexx messages, it *does not* provide for this. Thus, if an ARexx program is installed as a function host, it will close off access to all entries that follow it on the library list. Things that worked before the ARexx host was added to the library list will now break in mysterious ways.

"A function host must be able to notify the REXX process if it does not know about a function."

A similar problem arises if you forget to run a function host's executable or if you add the name of a nonexistent library to the list. In both cases ARexx will stop searching the library list at the point where it fails to find an entry, isolating entries that were added later or that have lower priority.

Despite these caveats it is instructive to implement an "almost" function host in ARexx to see how the rexxsupport.library functions openport(), closeport(), waitpkt(), getpkt(), getarg(), and reply() work. An example of such an almost function host, testhost.rexx, is included in the Weinstein drawer. Its companion, breakthings.rexx, demonstrates that things work well if you add this almost function host at a lower priority than all other items on the library list, but that things break if you add it ahead of another library. The general structure of the loops used in testhost.rexx to wait on the port and extract, process, and reply to waiting messages is the same as that used to write applications designed to handle messages coming from rexxarplib hosts. The program comments provide a detailed explanation of this strategy, as well as the way in which these function calls are used.

For the examples in the rexx subdirectory to function properly, you must make several additions to ARexx's library list. To automate this process I included the ARexx program addrexxlib.rexx. If you have not done so, you should now copy the contents of the usrc subdirectory to your C: directory (or elsewhere in your path) and copy the contents of the libs subdirectory to libs:. Finally, copy the contents of the rexx subdirectory to rexx:. Now you can open a CLI and type:

```
rx addrexxlib
```

Addrexxlib adds the necessary libraries to the library list and checks to see if the quicksort host is up and running. If it is not, addrexxlib launches the program and adds its port to the library list. Because I cannot live without the functionality provided by rexxarplib and rexxmathlib, I add the above command to my startup-sequence.

After running addrexxlib, execute the ARexx program breakthings.rexx. This example also serves as an introduction to the rexxarplib automatic requester facility accessed through the request() function. For additional examples of what the basic request() function can do, run the ARexx program reqsampler.rexx .

COMBINING A FUNCTION HOST AND REXXARPLIB

Quicksort is an example of a true function host. It is a simple host and only understands two commands, QSORT() and CLOSE(). Of these two commands only QSORT() is meant to be called from within an ARexx program. A call to the QSORT() function looks like:

```
call QSORT(begin,end,stem)
```

where begin and end are two integers, and stem is an ARexx stem variable. This function does a quicksort on the set of entries stem.begin, stem.(begin+1), ..., stem.end. Because QSORT() modifies the stem variable itself, if you need a copy of the original list it should be made before calling QSORT(). The CLOSE() command is used to close down the function host:

```
address 'QuickSortPort' CLOSE
```

QSORT() provides a complement to the rexxarplib filelist() function, which produces an unsorted listing of files and directories matching a given pattern. The filelist() function is one of the functions that depends upon having arp.library in your libs: directory. A full explanation of these functions is included in the body of the example, listfiles.rexx.

I have included several programs that show how to combine QSORT(), filelist(), and other rexxarplib functions to display a sorted listing of directories and files matching a given pattern. The first program is listfiles.rexx, which is meant to be run from a CLI. To see the difference between the way in which filelist() returns a list and what QSORT() does to it, run the program comparelists.rexx. Another reason for looking at this program is that it provides an example of using the address AREXX instruction to asynchronously launch a call to the request() function. This technique is necessary because comparelists.rexx opens two requesters to display the sorted and unsorted lists. Finally, to see how the listfiles.rexx program can be redone using the ARP file requester to get the directory, rather than a command line argument, see the program listfiles2.rexx. This last version of the program does not need to be launched from a CLI, making it suitable for installation in the fastmenu facility I'll discuss in the last section.

REXXARPLIB HOSTS

Although you can use automatic requesters, truly customizing your environment requires the ability to design your own requesters. To do this you must be able to open a window, on the Workbench or a custom screen, and endow it with text, graphics, menus, and gadgets. All this can be ac-

complished using the tools provided by rexxarplib, if you understand the concept of a rexxarplib host. For the remainder of this article, we'll examine this concept and a sample program that creates a rexxarplib host. (More extensive examples will follow in a future article.)

A rexxarplib host is an independently running process, launched by rexxarplib, whenever an ARexx program calls the createhost() function. To borrow from the terminology of object-oriented programming, each rexxarplib host is an *instance* of an object that knows how to open a window, perform various graphics operations in a window, add gadgets to a window, add menus to a window, handle gadget and menu events, and, if requested, pass along commands associated with menu selection or gadget clicks to another process. There is a direct parallel between the createhost() function and the NewObject command provided by all object-oriented programming languages.

Staying with our metaphor, any number of instances of rexxarplib hosts can be created, subject to memory limitations and the requirement that each one be uniquely identifiable. The general strategy for creating custom requesters is as follows:

- Create a uniquely identified rexxarplib host.
- Once the host is running, send it a message to open a window. As part of this message you specify the attributes that should be associated with this window—which IDCMP messages the host will receive from the operating system and whether the window will have a close gadget, drag bar, depth gadget, and so on.
- Once the window has opened, send messages to the rexxarplib host telling it to append imagery, text, menus, and gadgets of various types to the window. In part of these messages, specify an identification string for each gadget and the message to be returned each time the gadget is clicked. A similar process is followed for menus.
- Finally, if necessary, modify the port to which the results of various menu or gadget actions will be sent.

In general a call to createhost() looks like:

```
createhost(REXXARPHOST,REPORTPORT)
```

or:

```
createhost(REXXARPHOST,REPORTPORT,screen)
```

The first argument is the name of the port that the rexxarplib host opens, which uniquely identifies this rexxarplib host. The second is the name of the port to which it should forward the results of selecting a menu item, clicking on a gadget, and so on. The optional third argument is the name of the screen on which it should open. If a screen is specified, then it can be any screen opened by a call to the rexxarplib function OpenScreen(), or any public screen. Because version 1.3 of the operating system offers no support for public screens, this can only be the screen of an application that registers itself with screenshare.library (provided on the companion disk). Some such applications are VLT, TxEd Plus, and AmigaTeX preview. OS 2.0 does, however, support public screens. If you are running 2.0, your call to createhost() can include the name of any 2.0 public screen as well as the names of screens registered with screenshare.library.

In principle, the program that creates the rexxarplib host must open a port named REPORTPORT and wait for the messages from the rexxarplib host. There is, however, a way to avoid this: namely, to have the host forward all messages to ARexx or some other public message port. You do so via rexxarplib's SetNotify() function. The program that spawned the rexxarplib host then has no reason to wait around any longer, and it can exit. An example of this sort of program is fastmenu.rexx (in the Weinstein drawer).

FASTMENU: A SIMPLE EXAMPLE

This program creates a window endowed with menus and gadgets the user can click on to launch various applications. There are several companion files that are executed by selecting a menu item: assignlist.rexx, aliaslist.rexx, envlist.rexx, and help.rexx. The buttons, which are attached to the window managed by the rexxarplib host, launch Amiga executables such as Clock and Calculator, as well as some of the ARexx examples previously discussed. These commands are in the form of one-line ARexx programs, because all messages, except for CLOSEWINDOW, will be automatically forwarded to AREXX.

Note, there is a subtlety associated with the call to createhost(). If you look at fastmenu.rexx you will see that the call to createhost() actually looks like:

```
address AREXX " 'call createhost(FASTHOST,AREXX)' "
```

This is because createhost() does not return until the host it creates closes down. If you do not run this command asynchronously, the ARexx program hangs at this point. To avoid this, create an ARexx string program and use the address AREXX command to launch this program. Because address AREXX returns immediately, the original program continues on its way without interruption. Note that the first set of double quotes in this call is eaten when ARexx parses the line, and the second set of single quotes is just what the REXX process needs to identify this as a string program.

Finally, the call to SetNotify() looks like:

```
call SetNotify(FASTHOST,CLOSEWINDOW,FASTHOST)
```

This causes the CLOSEWINDOW message to be referred back to the port FASTHOST. Other IDCMP messages such as MENUPICK and GADGETUP, can also be redirected in this way. Redirecting CLOSEWINDOW back to FASTHOST works because every rexxarplib host understands a CLOSEWINDOW message and cleans up after itself. The example provided should allow you to generate our own customized fastmenu. I trust that once you do so you'll find it as indispensable as I do.

In future articles I will demonstrate how to use rexxarplib to create various useful utilities and provide a uniform interface to customize applications consisting of several independent programs. I will focus on rexxarplib.library because it is powerful, freely available, and defines a minimum standard that any such ARexx extension should meet. I hope that these examples will create an interest in other commercially supported packages of this type. We will all benefit immensely if developers turn their ingenuity to creating and documenting other utility libraries. ■

Marvin Weinstein uses ARexx and REXX extensively in his work at the Stanford Linear Accelerator. Write to him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (mweinstein).*

ON DISK

Global Parlor Tricks

Change the world with these geographic mapping algorithms.

By Howard C. Anderson

THE EARTH IS is a three-dimensional spheroid that may be represented by drawing the outlines of the land masses on the surface of a small sphere or globe. Globes, however, tend to be inconvenient to carry around. Flat pieces of paper that may be folded up are much more convenient. Unfortunately, it is impossible to flatten out a globe without stretching, compressing, twisting, tearing, or otherwise distorting it. The art of map making consists of devising schemes to transfer a drawing of the earth from a globe to a flat piece of paper while minimizing chosen types of distortion.

The results of these schemes are referred to as *map projections*. While the old fashioned method is to project a transparent globe's map onto paper and trace the outlines, calculating a few equations and rotation matrices on your Amiga is a much faster and more accurate approach. To help you visualize the mathematical concepts, let's discuss the tracing method first.

PROJECTIONS OVERVIEW

No one seems to be certain whether early map makers such as Mercator actually used a candle or lantern to project a map, but it is instructive to imagine how this would be done. If you draw a map of the world on a glass globe, place a tiny light source at the center of the globe, put a paper cylinder around the globe with its axis aligned with the axis of the globe, then the image of the map on the globe is projected onto the paper cylinder.You could then use a pen to trace the projected image on the paper. Figure 1 illustrates the central perspective projection. The globe here is centered on latitude = 0.0 and longitude = –100.0 degrees. The outlines of major features of the world are displayed in dark grey on the globe and their projections onto the cylinder wrapped around the globe are displayed in lighter grey.

By cutting the paper cylinder along a line parallel to its axis, it can be flattened out into a rectangular piece of paper. The map that results from such a procedure is called a *central-perspective cylindrical projection.*

The central-perspective projection is a member of a family of projections called cylindrical projections because you project the globe onto a cylinder. If you place the cylinder over the globe at some odd angle (not aligned with the earth's axis) the resulting map is known as an *oblique projection.* If the cylinder is aligned so that it is tangent to the north and south poles, the resulting map is known as a *transverse projection.*

Another family of map projections known as *conical projections* are created by using a paper cone instead of a cylinder. Imagine a paper cone being placed upon the glass globe like a hat and projecting and tracing the map upon it. The cone can then be cut along one side parallel to the axis. The result is a convenient, flat, paper map with distortions somewhat different than those created by cylindrical projections.

If you place a flat piece of paper tangent to the globe's surface and place the light source at the center of the globe, the resulting map is a *gnomonic projection.* If the light source is placed at the surface of the globe opposite the point of tangency of the paper and the globe, the resulting map is a *stereographic projection.* If the light source is placed opposite the point of tangency of the paper and the globe, and is located an infinite distance away, the resulting map is an *orthographic projection.* The orthographic projection shows the earth as it would appear from deep space. The gnomonic, stereographic, and orthographic projections are all *azimuthal projections* because the direction of a straight line drawn from the center of such a map to any other point on the map corresponds with the true direction or azimuth of that point.

Another type of projection is the *azimuthal equidistant projection* where the map provides the true azimuth and distance between the center of the map and any other point on the map. Such maps are not simple projections such as the ones previously mentioned. They cannot be produced by simple projection using a light source. These types of maps are still called "projections" but this term signifies a more general mathematical meaning of the word projection. The field of mathematics known as projective geometry defines this more general meaning.

There are many types of map projections that cannot be projected in a simple way by using a glass globe and a light source. Examples are: the *Mercator conformal projection,* the *Kavraiskiy projection,* the *Miller cylindrical projection,* the *Sinusoidal projection,* the *Mollweide projection,* and the *Van der Grinten projection.*

The Mercator projection is most closely related to the central-perspective cylindrical projection and is referred to as a cylindrical projection in most texts. In reality, it cannot be projected using a globe and a stationary light source. It is a mathematical projection specifically designed to force *rhumb* lines to be mapped as straight lines. A ship following a rhumb line never changes its heading. On the Mercator projection, a navigator can draw a straight line between where he is and where he wants to go, measure the angle, and use that as a constant heading. The trip takes a little longer because it is not a "great circle" route, but it is easy to steer. (Interestingly, if the navigator is not on a heading of due east or due west, and if he misses his target, and if he encounters no land, he eventually winds up at the north or south pole! Following a constant heading is the same as following a spiral path

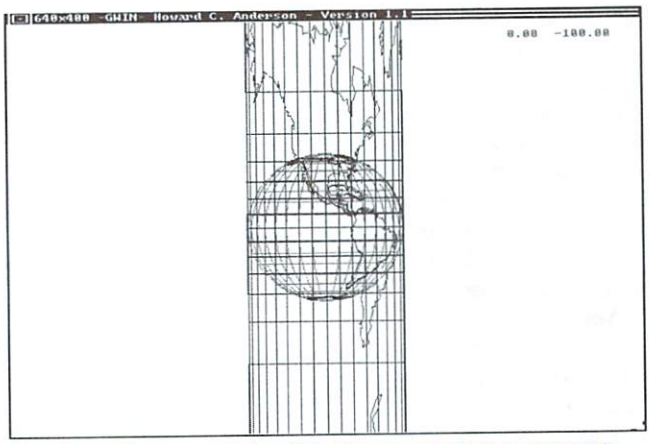


Figure 1: A central perspective projection with the globe centered at 0.0 latitude, -100 longitude.

known as a loxodromic spiral that terminates at a pole.)

Many of the "mathematical" projections are generated by construction rules involving a straight-edge and a compass. It is relatively easy to construct the latitude and longitude lines in this way. Skilled map-makers sometimes copy the contents from each latitude/longitude grid region from one type of projection to the corresponding latitude/longitude grid region of another type of projection and transform the shapes of the contents of these regions in their heads. If they are careful (and if the grid is small enough), they can transform, with reasonable accuracy, one type of map into another type of map in this manner.

In fact, as of the early 1980s, this is how *National Geographic* was producing their Van der Grinten maps. At that time I needed equations for the Van der Grinten projection. I checked all available references and found the straight-edge and compass method but no equations. I called the Defense Mapping Agency, and they didn't have the equations. I finally called *National Geographic* and found to my surprise that they were still using the straight-edge and compass method. The equations I needed, however, could be derived from the straight-edge and compass method. Time did not permit me to pursue the derivation. (Is it possible that these equations have still not been derived?)

Equations are known for most of the map projections in common use. These equations are relatively simple for projections centered at latitude = 0 and longitude = 0. The equations become very complex for projections centered on some other point.

MATHEMATICAL ROTATION OF THE GLOBE

When I began working on computer generated maps in 1978 I applied some relatively simple rotation matrix methods to the task. I was rather surprised by the results. The rotation matrices eliminated most of the complexity of the map transformation equations. The simplest form of the Mercator projection equations arises when the map is centered on latitude = 0 and longitude = 0. Use of rotation matrices allows the globe to be rotated so that the desired center point of the map (say latitude = 35, longitude = 23) is transformed to latitude = 0 and longitude = 0. All other points on the map are transformed similarly. The new set of latitudes and longitudes may then be run through the simple Mercator projection equations to produce the map. It is possible to cover the entire range of possible oblique Mercator projections using this method.

The same is true for orthographic projections: The simplest form of the orthographic projection arises when the map is centered on latitude = 0 and longitude = 0. By using rotation matrices to transform the desired center point of the map to latitude = 0 and longitude = 0 and similarly transforming all of the other latitudes and longitudes to their new values, the simplest form of the orthographic projection equations may be used to plot the map. This allows north polar, south polar, and all oblique orthographic projections to be done with relative ease. In fact, the rotation matrices can be used to simplify equidistant azimuthal projections and indeed all other types of projections.

The other important result is that it is relatively easy to derive the inverse transform of the map. The reason you need the inverse transform is that you might want to place the cursor on the map and read the latitude and longitude directly from the screen. The inverse transform tells you how to get the latitude and longitude from the screen cursor coordinates. The usual textbook version of the oblique Mercator projection equations is rather intimidating. Deriving the inverse would be no fun. However, it is easy to derive the inverse transform of a set of rotation matrices. It is also easy to derive the inverse transform of the simple form of the Mercator projection equations. This makes it simple to find the latitude and longitude from the cursor position. It is similarly easy for the orthographic and equidistant azimuthal projections. (I have not tried other types of projections but would expect them to be also simplified.)

THREE-SPACE ROTATION OPERATIONS

Rather than derive and describe the rotation matrices (which could be its own article), we will simply examine what is necessary to rotate the globe. For convenience, we will work with a globe of radius = 1.0 and three-space cartesian coordinates (x, y, z). We will also make five assumptions: The origin of the (x, y, z) coordinate system is at the center of the globe; The z axis points to the north pole; The x axis points directly toward you, the observer, and the y axis, perpendicular to the other two axes, points to the right.

We will also use three-space spherical coordinates (r, theta, phi). Theta is the longitude in degrees, phi is the latitude in degrees, and r is the distance from the center of the sphere to the surface. In all of the following operations, r is defined to be 1.0. If we need to calculate distances on the earth's surface, we may simply redefine r to be in kilometers or miles.

To rotate the globe, we perform the following steps:

1. Choose the latitude and longitude that will be in the center of the new map. Call this point (theta0, phi0). Let theta = theta0, and phi = phi0. ►

2. Convert the point (theta, phi) to a cartesian vector, (x, y ,z).
3. Rotate the (x, y, z) vector through an angle of –theta0 degrees about the z axis to obtain the (x', y', z') vector.
4. Rotate the (x', y', z') vector through an angle of –phi0 degrees about the y axis to obtain the (x", y", z") vector. We have now rotated the vector (x, y, z) corresponding to the latitude and longitude of the center point of the map so that the vector has coordinates (1, 0, 0) and points directly toward you, the observer.
5. Apply the same conversion and rotation to all latitude/longitude pairs making up the drawing of the earth. This assigns new vectors that correspond to new latitude and longitude values to each point of the drawing. We can then easily obtain the new latitude and longitude values from these new vectors.

We have effectively reassigned latitudes and longitudes to all points on the surface of the globe to what they would have been if early map makers had chosen the point (theta0, phi0) as the "true center" of the globe instead of the point they chose (the one that is on the equator and a little to the left of Africa.) We can then use these new values in the simplest forms of the map transformation equations to produce maps of the earth centered on the point (theta0, phi0).

The first task is to convert a latitude/longitude pair from (r, theta, phi) form to (x, y, z) form. The equations needed:

Equation Set 1

```
x = sin(pi/2 – pi*phi/180) * cos(pi*theta/180)
y = sin(pi/2 – pi*phi/180) * sin(pi*theta/180)
z = cos(pi/2 – pi*phi/180)
(where pi is 3.1415927...)
```

Now we have a vector, (x, y, z), from the center of the globe to the point on the surface of the globe specified by the latitude/longitude pair, (theta, phi). The next step is to rotate that vector about the z axis through an angle of –theta0 degrees. We use the matrix equation:

Equation 2

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos(pi*theta0/180) & \sin(pi*theta0/180) & 0 \\ -\sin(pi*theta0/180) & \cos(pi*theta0/180) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Next, to rotate this new vector about the y axis through an angle of –phi0 degrees, we perform the matrix equation:

Equation 3

$$\begin{bmatrix} x'' \\ y'' \\ z'' \end{bmatrix} = \begin{bmatrix} \cos(pi*phi0/180) & 0 & \sin(pi*phi0/180) \\ 0 & 1 & 0 \\ -\sin(pi*phi0/180) & 0 & \cos(pi*phi0/180) \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

The final phase is to map this new vector, (x", y", z") via the appropriate map projection equations.

THE ORTHOGRAPHIC PROJECTION

For the orthographic projection, we simply plot the (y", z") values whenever the x" value is positive. (When the x" value is positive, the vector denotes a point on the front side of the world. Ordinarily we will not want the back side of the world showing through.) We may wish to multiply the (y", z") values by some constant, C (a magnification factor), because they are at most one unit long. For example, if the screen resolution is 700x700, you may wish to plot the values (600y", 600z"). Most geographic map databases consist of a list of latitude and longitude pairs making up a line such as a coastline. There is usually an additional index number kept with each latitude and longitude pair that changes when you come to a new line. To draw a map, you move, using a nondrawing operation, to the first point of a new line and then draw to each successive point of the line until you reach a new index number. At that time, you move to the first point of this new line using a nondrawing operation, and continue the cycle. Figure 2 illustrates the orthographic projection obtained when the latitude and longitude of the map center point (phi0, theta0) are 45.0 and –100.0 degrees respectively. A south-polar projection, Figure 3 illustrates the orthographic projection obtained with a center point of latitude = –90.0, longitude = 0.0 degrees.

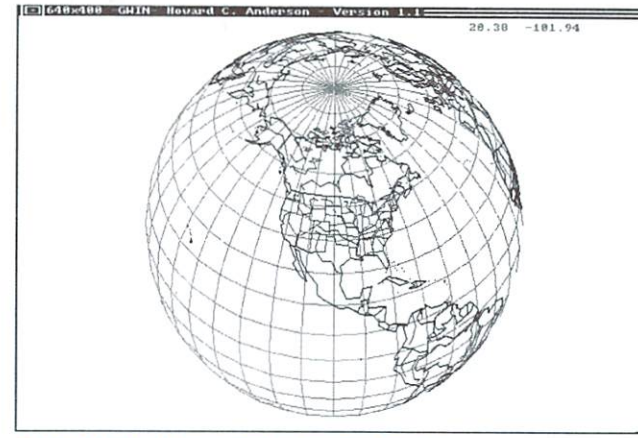


Figure 2: An orthographic projection with a map center at (45,–100).

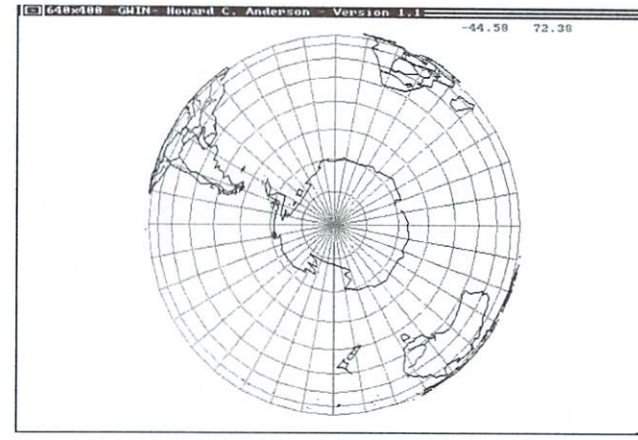


Figure 3: A south-polar orthographic projection.

THE MERCATOR PROJECTION

For the Mercator projection, we have:

```
xmerc" = (180/pi)*arctan(y"/x")
ymerc" = (180/pi)*log(tan(45+.5*Arcsin(z"))) for Arcsin(z") >= 0
ymerc" = –(180/pi)*log(tan(45–.5*Arcsin(z"))) for Arcsin(z") << 0
```

The xmerc" values are from –180 to +180, and the ymerc" values are from –infinity to +infinity. Mercator projections are always truncated in the vertical (+ymerc" and –ymerc") directions. Good software practice is to anticipate and eliminate the possibility of mathematical underflows and overflows. To do so, look at y" and x" before division (y"/x") and then

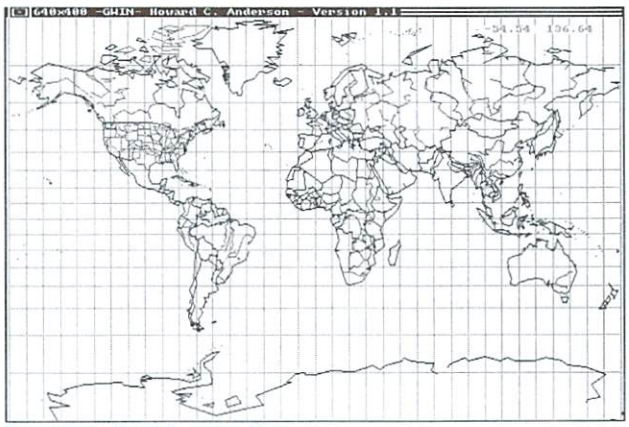


Figure 4: The familiar 0-latitude, 0-longitiude map.

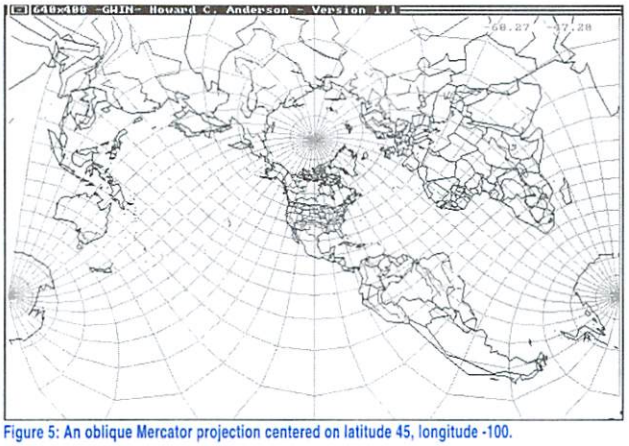


Figure 5: An oblique Mercator projection centered on latitude 45, longitude -100.

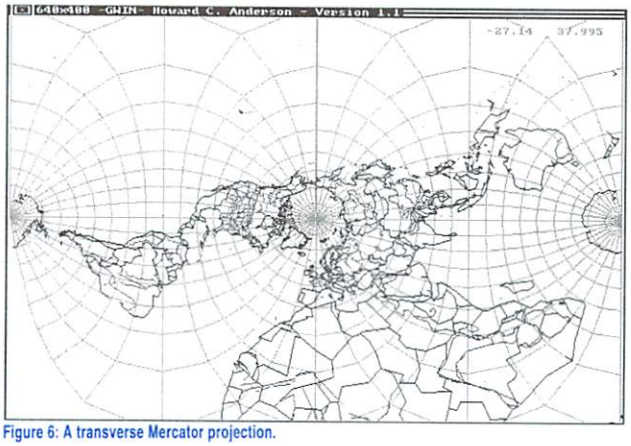


Figure 6: A transverse Mercator projection.

refuse to calculate or plot any extreme values. It is reasonable to limit the plotting region to be from –180 to 180 in the xmerc" direction and from –180 to 180 in the ymerc" direction. (In the software accompanying this article, the range in the ymerc" direction is restricted from –130 to +130 to compensate for the aspect ratio of the particular display screen used.) You may wish to multiply the (xmerc", ymerc") values by some magnification factor, C, to adjust the size of the map on the screen. The C*xmerc" and C*ymerc" values may then be plotted to produce a Mercator projection map.

Figure 4 is a Mercator projection centered on latitude = 0.0, longitude = 0.0 degrees. This is the map with which you are probably most familiar. An oblique Mercator projection, Figure 5 is a Mercator projection centered on latitude = 45.0, longitude = –100.0 degrees. Known as a transverse Mercator projection, Figure 6 is a Mercator projection centered on latitude = 90.0, longitude = 0.0 degrees.

INVERTING THE TRANSFORMATION

If we wish to read out the latitude and longitude by using a cursor on the map on the screen, we need to be able to convert the screen (xmerc", ymerc") location back to the latitude and longitude values corresponding to the map point displayed. This is now relatively simple; we merely reverse each transformation.

We first compute the vector (x", y", z") that corresponds to the cursor location. To do so for the Mercator projection, we first find the transformed latitude and longitude (theta", phi") from the screen coordinates (C*xmerc", C*ymerc"): Divide out C leaving (xmerc",ymerc"), then calculate:

```
theta" = (pi/180)*xmerc"
phi"   = pi – 2*arctan(exp((pi/180)*ymerc"))
```

Next compute the (x", y", z") vector:

```
x" = sin(phi")*cos(theta")
y" = sin(phi")*sin(theta")
z" = cos(phi")
```

So we now have (x", y", z") corresponding to the cursor designated position on a Mercator projection.

For the orthographic projection, the screen's cursor readout already provides (Cy", Cz"). After dividing out C to leave (y", x"), we must find x". From simple three-space geometry we see that:

```
x" = sqrt (1 – y"*y" – z"*z").
```

So we now have (x", y", z") corresponding to the cursor designated position on an orthographic projection.

From the (x", y", z") vector obtained by either process, we can determine the latitude and longitude in original earth coordinates by the following process:

1. Reverse the rotation about the y axis by rotating through an angle of +phi0 degrees:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos(pi*phi0/180) & 0 & -\sin(pi*phi0/180) \\ 0 & 1 & 0 \\ \sin(pi*phi0/180) & 0 & \cos(pi*phi0/180) \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

2. Reverse the rotation about the z axis by rotating through an angle of +theta0 degrees:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos(pi*theta0/180) & -\sin(pi*theta0/180) & 0 \\ \sin(pi*theta0/180) & \cos(pi*theta0/180) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

The resulting (x, y, z) vector is the vector corresponding to the cursor designated position in the original world coordinate system.

3. Determine the latitude and longitude with:

```
latitude  = 90 – (180/pi)*Arccos(z) degrees
longitude = (180/pi)*Arctan(y/x) degrees.
```

PRACTICAL CONSIDERATIONS

In practice we can increase speed considerably by premultipling, precomputing, and storing the rotation matrices ►

so that only simple multiplication operations are required for the rotations. This increases the speed considerably. The example C procedures in the accompanying disk's Anderson drawer demonstrate this technique.

Note that we have not taken the oblateness of the earth into account. Viewed from a point over the equator in deep space, the outline of the earth appears to be an ellipse rather than a circle. Assuming that latitudes and longitudes are assigned to points on the surface of the earth to be the angles associated with a ray from the center of the earth, it is evident that certain map projections will have an error associated with this ellipticity. For example, the Mercator projection will be slightly incorrect with respect to latitude. Dutton's *Navigation & Piloting* says:

"Since these variations from a truly spherical shape are so slight, for most navigational purposes the earth can be considered a sphere, and solutions of navigational problems based on this assumption are of practical accuracy. In the making of charts, however, consideration is given to the oblateness of the earth."

To account for oblateness in the Mercator projection ymerc" value shown above simply add the correction term:

.5*E ln((1–E*|z"|) / (1+E*|z"|)) (where E is the eccentricity of the earth)

DRAWING CIRCLES ON A MAP

Over ten years ago I devised a "trick" for drawing circles on the globe centered on any point so that the circles are accurately displayed (distorted) under any projection. The trick is to calculate the latitudes and longitudes of a circle of the appropriate radius centered on the north pole. You then create a new set of simple rotation matrices to carry the north pole to the desired center point of the circle on the map. The points of the circle are then run through this new rotation matrix. We run those results through the same rotation matrix and projection equations that the original map points were run through. When the results are plotted, you have a circle of the appropriate radius centered on the desired location. The circle will be displayed and distorted appropriately for the particular map with which you are working.

To begin this process you first find the latitude of the circle you wish to map by applying a simple equation from high school geometry, S = R * phi. S is the length of an arc, R is the radius of a circle, and phi is the angle between the two radius vectors that define the arc. If we let R be the radius of the earth and S be the radius of the circle we wish to draw on the map, we can calculate the angle phi as:

phi = S/R

If we choose one of the radius vectors to be aligned with the north pole, we see that the latitude of the desired circle about the pole is 90 – phi. To save this circle, we simply list and save the latitudes and longitudes of the points around the circle. The latitude is the same for all point pairs, i.e., 90 – phi. For the longitudes, we start at –180 and count up by any desired increment until reaching +180.

Now we need to construct rotation matrices that will rotate the center of the circle (the north pole) to the desired center point of the circle on the map. We will use these rotation matrices to rotate the points of the circle to their correct latitude and longitude values (the values that they would have on an unrotated map.) Let's assume that the circle is to be drawn centered on latitude = lat0 and longitude = lon0.

As above, we first convert the latitude/longitude pair from (r, theta, phi) form to (x", y", z") form. The equations are:

Equation Set 4

x" = sin(pi/2 – pi*theta/180) * cos(pi*phi/180)
y" = sin(pi/2 – pi*theta/180) * sin(pi*phi/180)
z" = cos(pi/2 – pi*theta/180)
(where pi is 3.1415927...)

In this case, we want to rotate the point (r, 0, 90), the north pole, to (r, lon0, phi0).

First, rotate (x", y", z") about the y axis through an angle of –(90-lat0) or (lat0–90) degrees. The matrix equation necessary:

Equation 5

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos(pi*(lat0-90)/180) & 0 & -\sin(pi*(lat0-90)/180) \\ 0 & 1 & 0 \\ \sin(pi*(lat0-90)/180) & 0 & \cos(pi*(lat0-90)/180) \end{bmatrix} \begin{bmatrix} x" \\ y" \\ z" \end{bmatrix}$$

Next, rotate the new vector (x', y', z') about the z axis through an angle of +lon0 degrees. The matrix equation to perform this rotation is:Now apply equations 4, 5, and 6 to

Equation 6

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos(pi*lon0/180) & -\sin(pi*lon0/180) & 0 \\ \sin(pi*lon0/180) & \cos(pi*lon0/180) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x" \\ y" \\ z" \end{bmatrix}$$

each latitude/longitude pair representing the circle that we saved. The resulting (x,y,z) vectors are then transformed via the map projection equations 1 and 2 and plotted using either the Mercator projection equations or the orthographic projection equations as shown above.

Figure 7 shows circles of various centers and radii drawn accurately on an orthographic projection centered on latitude = 45.0, longitude = –100.0.

Figure 8 shows circles of various centers and radii drawn accurately on a Mercator projection centered on latitude = 45.0, longitude = –100.0. Note that on a Mercator projection, circles are always displayed as circles— they maintain their shape. This is due to the *conformal* or *shape preserving* property of the Mercator projection. The area of a displayed shape

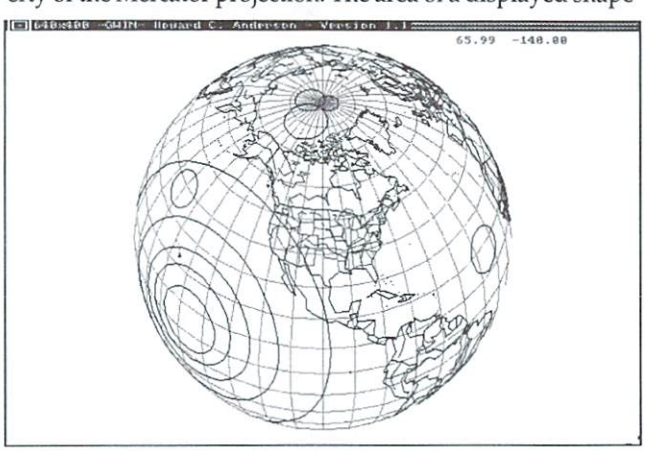


Figure 7: Various accurately drawn circles on an orthographic projection.

is distorted however so that a circle of given radius appears larger when it is further vertically from the map center.

WORLD MAP DATABASES

To create your own maps, you need latitude/longitude pairs. The best source for these are map databases. Perhaps the most well-known world-map database is World DataBank II that was developed by the CIA presumably by digitizing satellite photos. World DataBank II contains nearly six million latitude/longitude pairs. Additional data accompanies each latitude/longitude pair—the line-segment number, the rank or type of data, and the number of points in the line segment. Plotting such data is a simple matter of transforming each latitude/longitude pair, lifting the pen whenever the line-segment number changes and leaving the pen down between successive points when the line-segment number remains the same. You could also change pen color whenever the rank changed. World DataBank II contains 27 ranks or categories of data. The types of data are as follows:

```
I. International boundaries or limits of sovereignty
   01 Demarcated or delimited
   02 Indefinite or in dispute
   03 Other line of separation or sovereignty on land
II. Coastlines, islands, and lakes
   01 Coastlines, islands and lakes that appear on all maps
   02 Additional major islands and lakes
   03 Intermediate islands and lakes
   04 Minor islands and lakes
   06 Intermittent major lakes
   07 Intermittent minor lakes
   08 Reefs
   09 Major salt pans
   10 Minor salt pans
   13 Major ice shelves
   14 Minor ice shelves
   15 Glaciers
III Rivers
   01 Permanent major rivers
   02 Permanent minor rivers
   03 Additional rivers
   04 Minor rivers
   05 Double-lined rivers
   06 Major intermittent rivers
   07 Additional intermittent rivers
   08 Minor intermittent rivers
   10 Major canals
   11 Canals of lesser importance
   12 Irrigation canals
IV. Internal boundaries
   01 First order administrative
```

World DataBank II is available from the National Technical Information Center, which is part of the Commerce Department. A version of World DataBank II is available on Internet, as well. The Internet system maintains archives of programs and data that can be readily retrieved. The listing for the directory containing World DataBank II follows:

```
/usr/spool/ftp/tmp/WorldMap:
total 3041
-rw-r—r— 1 swo     3300000 Jan 13 00:40 Map.tar.Z.1
-rw-r—r— 1 swo     3300000 Jan 13 00:40 Map.tar.Z.2
-rw-r—r— 1 swo     3045353 Jan 13 00:41 Map.tar.Z.3
```

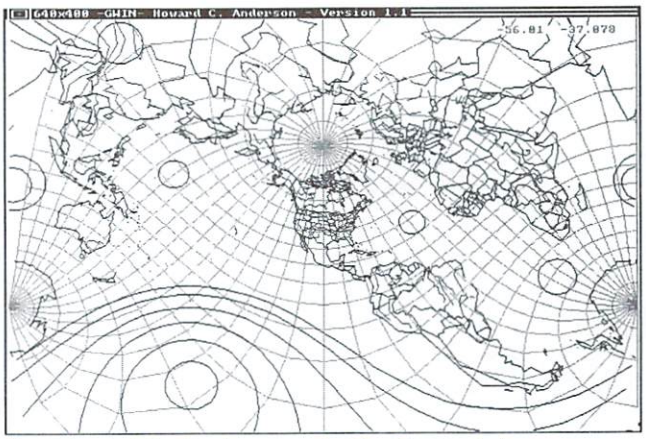


Figure 8: Accurate circles with various centers and radii on a Mercator projection.

```
-rw-r—r— 1 swo     2705037 Jan 23 23:16 Map_small.tar.Z
-rw-r—r— 1 swo        1431 Jan 23 23:21 README
```

You can retrieve the Map.tar.Z.1 file, for example, using the UNIX command below:

```
uucp -r uunet\!/usr/spool/ftp/tmp/WorldMap/Map.tar.Z.1  local_path
```

in which local_path is the full pathname of the local file in which you wish to store the data. (I advise you to seek help locally if you are not experienced with Internet uucp usage.) Make sure you set aside enough space (and time) to download; the WorldMap directory contains nearly 12.4 megabytes of data. Also note that the data is in compressed binary form. The usable uncompressed form is considerably larger.

The map database I used for my mapping routines came from Fred Fish disk #262. The three map files in the disk's WorldDataBank directory appear to be extracts of various resolutions from World DataBank II. The map files consist of short integer x and y values apparently obtained from a Miller projection followed by application of an aspect ratio correction factor (the documentation doesn't say for sure). I transformed (by inverting the Miller transformation) the values in the files back to real latitude and longitude values for use with my mapping routines. The precision of each value is terrible of course, because the designers of this database used short integers for data storage to speed up their graphics plotting routines which are, incidentally, extremely fast.

The values I obtained from the Fish databases were sufficient, however, for demonstrating my map transformation equations. You may notice that Greenland seems to have been damaged somehow. It is flat on top and does not reach its proper latitude. The other continents, including Antarctica, seem to be fine. (It is possible that I am wrong about the original projection having been a Miller projection.) Each file also contains information regarding data type and segment numbers so that your program knows when to lift the pen and when to change colors to signify different types of map data.

PROGRAM LISTING

Geomap (in the Anderson drawer) shows the techniques we've discussed in action. For clarification, you will find extensive documentation regarding the graphics system calls on Fred Fish disk #322. You should be able to use most of the internal procedures, however, without knowing the details of the particular graphics system used here. I attempted to isolate the map-data handling procedures and transforma-

Continued on p. 39

Multitasking in Amiga Basic

The key is to do nothing, efficiently.

By Robert D'Asto

ALTHOUGH SELDOM MENTIONED in articles and texts, Amiga Basic lets you create applications that take advantage of the Amiga's most touted feature. In fact, the ability to coexist in the system's multitasking environment is built into the language and writing such programs can be surprisingly easy.

THE AMIGA TASK FORCE

Starting a program on a single-tasking computer is usually accomplished by directing the machine's central processing unit to the memory location of that program's machine code instructions and letting the CPU execute them until the program ends or is stopped. While that program runs, the CPU knows nothing of any other programs that may exist in memory and is not capable of executing any other application without ceasing execution of the first. You can switch back and forth on a single-tasking machine, but while one program is being employed by the user the others pause and accomplish nothing.

The simultaneous execution of two or more separate applications requires that each program observe certain "rules of the road" to avoid conflicts. Two programs attempting to write to the same file at the same time, for example, could produce an unreadable mess or even a system crash. Also, programs that hog the central processing unit or other system resources can slow down or completely stop concurrent applications from their appointed tasks.

When an Amiga program is started, it does not simply run—that is, the CPU is not directed to its machine code instructions for execution. Rather, Amiga programs are said to be launched. A portion of the operating system, known as Exec, is informed that a new set of instructions is to be added to those already executing, and this new program is then incorporated into the system's current task list, a system record of currently running programs. Exec then sees to the orderly sharing of the CPU and system resources by that program and those already running. The new program is then started, with Exec monitoring and directing the execution of all current tasks. At any given instant the CPU is executing only a single instruction of a single program, but like a traffic cop at a busy intersection, Exec orchestrates the flow of each program's instructions to the CPU for orderly execution in turn. When run in the CPU's microsecond time frame, the result is simultaneous execution of multiple tasks.

There are actually two types of Amiga multitasking programs: a task and a process. Their differences have little application for BASIC programmers, but you may run across the terms when studying more advanced Amiga issues, so an explanation is in order. Tasks and processes are both multitasking-capable applications. Simply put, the primary difference between the two is that a task cannot perform input/output by using AmigaDOS. A task could not, for example, read or write to a disk file, at least not through normal means. A process does not have this limitation and can perform any Amiga operation a programmer is capable of writing. Amiga Basic programs run as processes, as do most Amiga applications. The term "task" is also used (as it is here) to denote either a task or a process—any application designed to run in harmony with the Amiga's multitasking environment. An application that is neither a task nor process would be one that is not written to function within the conventional Amiga system. Games that bypass the operating system and command the undivided attention of the CPU are good examples of the latter.

MULTITASKING ETIQUETTE

Writing applications intended for multitasking use requires the observance of certain programming guidelines. As stated above, Amiga multitasking works through a system of sharing the CPU, that is monitored by the Exec. Each program has a turn at using the CPU to execute some portion of its instructions. This may seem to imply that when two programs are running, for example, each will run at half its normal speed, but this is not the case in practice. While it is true that multiple tasks running simultaneously often run slower than a single one, the slowdown is normally far less than a direct ratio of the number of running tasks. This is because most applications spend the majority of their running time waiting for the user to do something such as press a key, operate the mouse, select an option, and so on. Programs that are written in observance of the Amiga's multitasking protocols relinquish their demands on the CPU during these waits, allowing concurrent tasks the opportunity to use it.

A frequently encountered BASIC device, the "busy loop" should be avoided in any program intended for a multitasking environment. For example:

```
WHILE INKEY$="":WEND
```

is often used to suspend a program until the user presses a key. This loop continually checks for any character entered from the keyboard and, if none is detected, repeats itself. This is fine for use during program development when you need a quick pause device, but including such a loop in a finished application will hog the CPU, inhibiting or preventing concurrent tasks from using it to execute their own instructions.

Another often-seen BASIC device that should be avoided is the "pause loop." This is usually written as:

```
FOR x=0 TO 5000:NEXT
```

and is intended to pause execution for some interval. This has the same drawback as the busy loop. A more multitasking-friendly alternative to both of these devices is given in the example below (and in the DAsto drawer on disk).

THE BIG SLEEP

Although sparsely documented in the Amiga Basic manual, the SLEEP command is BASIC's built-in key to writing multitasking applications. The Amiga Basic manual states that SLEEP "causes [an] Amiga Basic program to temporarily suspend execution until an event occurs that Amiga Basic is interested in, such as a mouse click, key press, object collision, menu select, or a timer event." Actually, this is not the whole story. According to the description, using the SLEEP command alone should cause the program to suspend execution at that point until one of these events occurs, and then continue execution with the next line of code. This, however, is not the case. Including the SLEEP command in a program produces no pause at all. The program runs as before, as if the command had no effect.

How, then, is the SLEEP keyword used to suspend execution of a program? One way is by executing SLEEP *twice* in succession. Using SLEEP:SLEEP *will* cause the program to pause until a key is pressed, the mouse button is clicked, or any of the events given in the SLEEP description occur. You can easily demonstrate this by typing the single command, SLEEP, into the editor and running this as a program—nothing happens. Now, add a second SLEEP command and run it again. This time the program starts and then pauses until you press a key or click the mouse, at which time the program ends and returns to the editor. The pause produced in this way is a multitasking-friendly suspension of program execution that, unlike the busy loop, will not interfere with the execution of concurrently running system tasks.

This, however, raises another question. Because at least five different events can wake up a sleeping program, how will the program know which one occurred? This is where Amiga Basic's event-trapping feature comes in. All of the events that trigger SLEEP into continuing execution can be trapped with the

```
ON event GOSUB label
```

command format that is described in Chapter 6 of the Amiga Basic manual. Before invoking SLEEP, the program should establish where execution will branch when a particular event occurs. Most programs will not use all of the possible event-trapping features, so some device must be used for ignoring those events that the program is not interested in. This can be done with a SLEEP loop, which is any loop that includes a single invocation of the SLEEP keyword. The program first establishes which events are of interest and then enters the SLEEP loop to wait. For example:

```
ON MOUSE GOSUB MouseRoutine
MOUSE ON
Loop:SLEEP:GOTO Loop
```

This causes the program to stay within the loop until a mouse click occurs, at which time a subroutine called MouseRoutine is executed. When this subroutine finishes, the program returns to the loop and awaits another click. Using SLEEP within the loop allows concurrently running tasks to continue without undo interference from your program; the Amiga's multitasking protocols are maintained by SLEEP. You can add additional events of interest, each specified in the format:

```
ON event GOSUB label:event ON
```

Remember that SLEEP should be used in every short, repeating loop contained in your listing if you want the program to function harmoniously with other tasks. What about longer loops or loops that repeat only a few times? If in doubt, use SLEEP. You can always check if the command is really needed by later omitting it, then running, and testing the program.

The following program illustrates the use of SLEEP and event-trapping for creating a multitasking "background task" in BASIC. This program opens a tiny window in the upper-left corner of the Workbench screen that displays the current system time and updates this display once per minute without interfering with other applications in use. Double-click its icon in the DAsto drawer to launch the task. (When writing your own routines, don't forget to save the program first and then start it by clicking its icon, as starting such programs directly from the editor can have unpredictable results.)

```
WINDOW 1,"",(0,11)-(40,27),16+8,-1
GOSUB ShowTime
ON TIMER (60) GOSUB ShowTime
TIMER ON
WHILE WINDOW(7)
   SLEEP
WEND
SYSTEM
ShowTime:
   WINDOW OUTPUT 1
   PRINT
   PRINT LEFT$(TIME$,5)
RETURN
```

While this clock program continues to run, other programs can be used within the Workbench environment without noticeable interference from the clock. Because it actually does something only once per minute, other tasks are free to use the CPU during the waiting interval. You can terminate the program at any time by clicking the window's close gadget.

Reviewing the code, SLEEP is used in a WHILE/WEND loop that calls the WINDOW(7) function to check for the existence of the window. If the window has been closed by clicking on its close gadget, WINDOW(7) returns zero and the loop exits, causing SYSTEM to execute and the program to

Continued on p. 62

TurboText v1.02 Professional Text Editor

The any-way-you-want-it text editor.

By Tim Grantham

EVERYTHING ABOUT THIS fine programmable text editor says craftsmanship. You will first see it in TurboText's careful adherence to the look-and-feel of AmigaOS 2.0. All the program's requesters employ the same, three-dimensional appearance. TurboText (hereafter called TTX) uses the system font requester to let you choose the text font. It also automatically adopts the same font, size, and colors as the Workbench screen. (You can, of course, specify separate preferences for TTX when using a custom screen.)

TTX goes beyond mere cosmetics, however, to take advantage of more advanced 2.0 features. When running on the Workbench, TTX's AppWindow support lets you simply drag an icon into a TTX window to load the matching file. When running on a custom screen, TTX opens a public screen, allowing other applications to open windows on the TTX screen. (This welcome feature lets me open a Shell window on the TTX screen; see Figure 1.) TTX can open on a virtual screen of up to 998×998 pixels and use the 2.0 autoscroll feature to display hidden ranges, as well.

SERVER AND CLIENT

In addition, TTX provides highly integrated support for ARexx. In fact, TTX might best be described as a text editing server, providing 144 different text manipulation commands to any client application that can speak ARexx.

A TTX window could also be described as a client. Each one calls the same TTX commands used by external applications. To see how this works, take a look at the definition files used to create the user interfaces for TTX. Each file consists of the names of keystrokes, mouse buttons, menu items, and the TTX command each one invokes. Even the gadget labels used in TTX requesters can be changed in the definition files. The definition files supplied with TTX provide emulations of several popular text editors, including the Amiga's native Ed and Memacs, MicroSmith's TxEd, and ASDG's CygnusED Professional, plus QEdit, WordStar, and BRIEF from the MS-DOS world. The files also provide French and German versions of the standard TTX menus.

Each definition file can also contain a user-defined dictionary and templates. The dictionary enables you to correct misspellings by striking the F1 key while the cursor is on or next to the word. The text editor then uses a best-fit algorithm to replace the misspelled word with one from the dictionary. Templates extend this capability by letting you generate commonly used pieces of text with just a few keystrokes. I found templates especially useful for quick creation of "blank" Intuition data structures, complete with comments. TTX comes equipped with definition files containing dictionaries and templates for C, 68000 assembly language, Ada, Cobol, and Modula 2.

FLEXIBILITY, PLUS

Several other inventive features elevate TurboText above the rest of the text-editor crowd:

• Folds—TurboText uses this key feature of outline processors to collapse or hide a series of lines into a single line or heading. I find it very useful for listings containing reams of data structures. Each one can be folded so that only the name of the structure is displayed. A character in the first column indicates the fold. A keystroke will unfold the lines to reveal the contents; another folds it back up again. Nested folding is also possible. The file can be saved with the folds intact by using the following nifty feature.

• Icon commands—When I first configured TTX, I turned off the saving of project icons for my TTX files. After all, I work mostly from the CLI, rather than the Workbench—what do I need icons for? TTX, however, can store nontextual information contained in a file in the icon for that file. This enables it to retain the location of folds and bookmarks, while still providing a pure ASCII file for the compiler. Martin Taillefer, the creator of TTX, has extended this concept to enable the icon to contain the tab width used in the document and the last change made to it. I now save icons with my files. For a next step, I'd like to see the inclusion of preference and configuration filenames in the icons, so that a document's entire environment could be loaded with it.

• Templates for file backups—The backup template provides control over the name and location of the backup file. I use this to automatically create a version history of my program, using the template MyProg#.c. When creating the backup, TTX first checks the directory for the matching filename and then substitutes a number one higher for the # character in the template, as in MyProg6.c. TTX also lets you set the upper limit it may use for the version number before starting again at zero.

• Hex editing window—The hex window opens a second editable view on a file, similar to that displayed by the TYPE filename hex AmigaDOS command. This eases the editing of binary files, in particular. The hex editing window is completely synchronized to the original window, with text changes and cursor movement shown in both windows simultaneously. If you open an Information

window for the file, which dynamically displays such statistics as file size, average characters per line and available memory, you can provide your friends with an effective demonstration of intertask communication on the Amiga.

• Macro record mode—Other text editors provide a record mode to create macros. TTX, however, also stores each macro as a series of TTX ARexx commands, greatly simplifying the process of creating more complex ARexx client programs. Such programs can also be launched from within TurboText. TTX comes with almost 30 macros that, besides demonstrating how to control TTX through ARexx, provide useful functions.

• Programmer's calculator—This separate program can be brought up on either the Workbench screen or TTX's public screen. Operable via either the mouse or keyboard, it provides arithmetic and logical functions for binary, octal, decimal, and hex numbers.

Rest assured, TTX has many of the other features that we have come to expect in Amiga text editors, including vertical block operations, hot-key startup, status lines, multiple views, fast scrolling, and wildcard support in the file requester. TTX can open as many files as memory permits and supports the three-button mouse, the extended Amiga keyboard, and the Amiga clipboard device.

Be warned, however, that TTX's printing capabilities are strictly limited to those provided by the Amiga's printer preferences.

A WISH LIST FOR PERFECTION

In future versions of TurboText I would like to be able to assign different colors within the available palette to the text, background, border, and title of each window, as I can currently with TransWrite (Gold Disk). Other features worth adding include: the option to store text-style data in the icon, just as fold locations are stored; the use of wildcard characters in word searches; and support for multiple selection in the file requester.

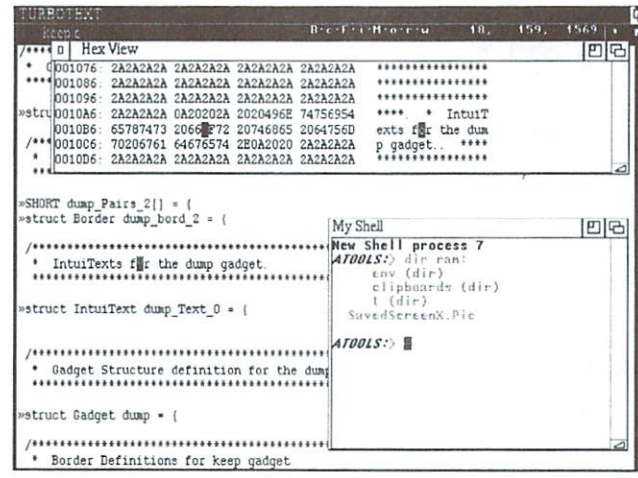


Figure 1: TurboText lets you open windows onto its window.

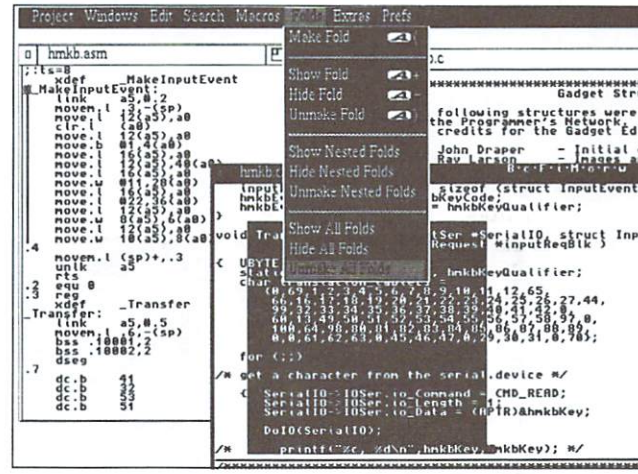


Figure 2: A close-up of a 998x998 screen.

The only problem I encountered while using TTX turned out to be a bug in the version of the operating system (Kickstart 37.92 and Workbench 37.33) I was running on my Amiga 3000. If I changed the font used to display file text while the console window was open on the TTX screen, all the windows would immediately close without giving me the opportunity to save their contents. This would also suspend the operation of TTX—not even the hot-key startup would get the program going again. I subsequently ran the same copy of TTX under a later version of the operating system (Kickstart 37.175 and Workbench 37.55), and it performed flawlessly.

The manual matches the program's exemplary design and execution. It's well-written and thoughtfully organized, with an excellent TTX command reference for programmers. It also includes an adequate glossary and index. The screengrab halftones used to illustrate the manual are too coarse, however; they should have been output at a finer screen resolution. The manual also refers to a DONOTWAIT tool type but provides no explanation of it.

THE VERDICT

TTX serves as a model 2.0 application in every respect, yet runs almost as well under 1.3. The manual carefully delineates the differences in operation. As you may have surmised, TurboText v1.02 is now my preferred text editor. I unreservedly recommend it and rate it 9.5 out of a perfect 10. ■

TurboText v1.02
Oxxi Inc.
PO Box 90309
Long Beach, CA 90809-0309
213/427-1227
$99
No special requirements.

TNT

Technical News and Tools from the Amiga Community.

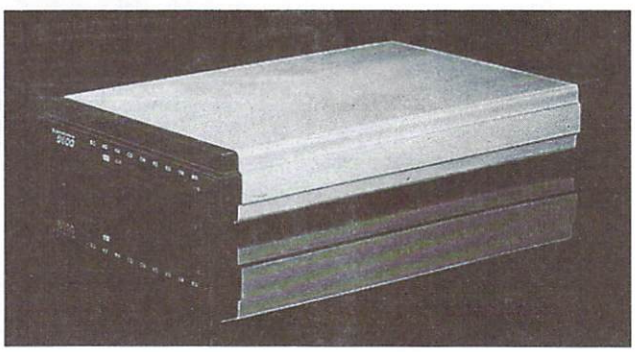
Compiled by Linda Barrett Laflamme

New Identity

The name will change, but the products remain the same. Great Valley Products (600 Clark Ave., King of Prussia, PA 19406) has acquired the entire Lake Forest Logic product line. GVP will now distribute and support **A.D.A.P.T. The 680×0 Assembler**, **The Disk Mechanic**, and **Macro Paint**. For technical support of these products, you can either log on to GVP's BBS (215/337-5815) or call a tech support representative (215/337-8770).

Faster, Faster

Supra Corporation has stepped up the on-line pace with the **SupraModem 9600** ($699.95). The modem features CCITT V.32, CCITT V.42bis, and MNP 2–5 support allowing transfer rates of up to 38,400 bps. Backed by a five-year warranty, the modem offers Hayes compatibility, asynchronous and synchronous operation, autoanswer/autodial (tone or pulse), two phone jacks, and an adjustable-volume speaker. You supply the software and RS-232C cable. For a full description, contact Supra Corp., 1133 Commercial Way, Albany, OR 97321, 503/967-9075.



Supra's 9600 baud SupraModem.

Drivers and Libraries

Need sound for your Amiga or CDTV software? Try the **MaxTrax** music driver.

Record your music in real time from MIDI keyboards, import it from standard MIDI or SMUS files, or enter it directly from Music-X (included with MaxTrax). When you are satisfied with the sound, simply save the music in MaxTrax performance format, ready to be played from your program.

MaxTrax uses standard Amiga sound generators, with up to 16 dynamically assigned tracks. The dynamic volume control allows global fade-ins and fade-outs under host control, while the channel volume control gives composers full control over music dynamics. A wide range of of musical effects are supported—accelerando/decelerando, MIDI Pan support, attack velocity, pitch bend, and portamento. Synchronization events allow synchronization of visual and other program events with specific points in the music, allowing you to create cuts, wipes, and other transitions synchronized to your compositions. Plus, your host program can freely insert sound effects into the music stream.

Written in assembly, MaxTrax is interrupt-speed nonspecific and works with both NTSC and PAL displays.

The MaxTrax Software Developer's Package ($500) includes the Music-X MIDI sequencing package, MaxTrax driver object modules (both Manx C and SAS C formats), utility programs, sample code and songs, IFF sampled instrument sounds, developer documentation, and technical support. Each MaxTrax Product Insertion License is $2000. For a demo disk, send $10 to The Dreamer's Guild, PO Box 33031, Granada Hills, CA 91394-0931, 818/349-7339.

The Dreamer's Guild also offers the **GT Emulator**, a GadTools emulation library that recreates all function calls in the 2.0 GadTools library. With it you can program all the new 2.0 features and remain compatible with 1.3.

GT Emulator operates transparently to the user and includes intelligent menu layout functions and an automatic Palette Snooper that can create the beveled 2.0 look under any 1.3 palette. All the standard 2.0 gadgets (buttons, list views, cycle gadgets, mutually exclusive sets, scrollers, and the palette gadget) can be created.

For $2000 the GT Emulator Development Package comes complete with the library, programming instructions, and a site license for unrestricted, royalty-free use within any of your company's products. ►

The AmigaWorld Tech Journal Disk

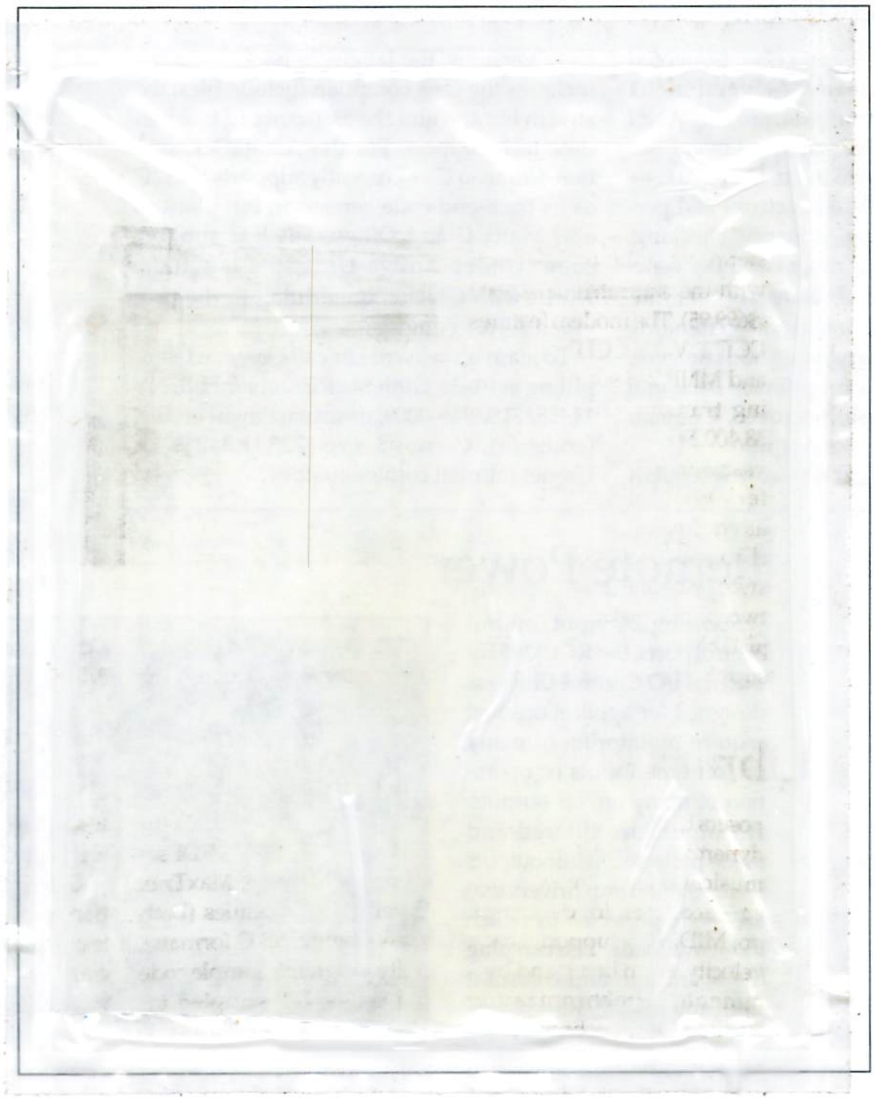
ON DISK

In addition to source and executables for the article examples, you'll find the lastest versions of:

A68k Assembler—A fully functional 68000 assembler

BLINK—The Software Distillery's linker

BlitLab—A safe environment for testing your Blitter ideas



This nonbootable disk is divided into two main directories, *Articles* and *Applications*. Articles is organized into subdirectories containing source and executable for all routines and programs discussed in this issue's articles. Rather than condense article titles into cryptic icon names, we named the subdirectories after their associated authors. So, if you want the listing for "101 Methods of Bubble Sorting in BASIC," by Chuck Nicholas, just look for Nicholas, not 101MOBSIB. The remainder of the disk, Applications, is composed of directories containing various programs we thought you'd find helpful. Keep your copies of Arc, Lharc, and Zoo handy; space constraints may have forced us to compress a few files.

Unless otherwise noted in their documentation, the supplied files are freely distributable. Read the fine print carefully, and do not under any circumstances resell them. Do be polite and appreciative: Send the authors shareware contributions if they request it and you like their programs.

Before you rush to your Amiga and pop your disk in, make a copy and store the original in a safe place. Listings provided on-disk are a boon until the disk gets corrupted. Please take a minute now to save yourself hours of frustration later.

If your disk is defective, return to AmigaWorld Tech Journal Disk, Special Products, 80 Elm St., Peterborough, NH 03458 for a replacement.

Laser Programming

Written by Andrew Binstock, David P. Babcock, and Marv Luse, ***HP LaserJet Programming*** aims to teach C programmers how to write applications and tools for Hewlett-Packard LaserJet II and LaserJet III printers. Through discussion and programming examples, the book examines the LaserJet's PCL language, fonts, font manipulation, raster-graphics generation , translation of PCL to English or other printer commands, and the processing and justification of text. Weighing in at 432 pages, *HP LaserJet Programming* sells for $26.95 and is published by Addison-Wesley Publishing Company, Reading, MA 01867, 617/944-3700.

An OOP from CC

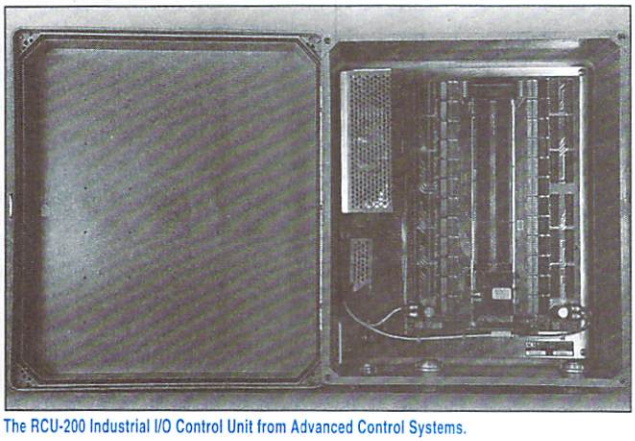
Ready to make the leap to object-oriented programming? **Comeau C++ version 2.1** promises to ease the transition, offering ANSI C compatibility that lets you port over your old C code without breaking it. Comeau C++ is a licensed port of AT&T's cfront and performs syntax checking, semantic checking, error checking, and all other compiler functions. Input C++ code is translated into internal compiler trees. Instead of generating an internal proprietary intermediate code from these trees for a proprietary back-end code generator to use, however, Comeau C++ generates C code for output.

Available in AmigaDOS and A3000UX Unix versions, the language package ($250) includes the C++ compiler, include files, the stream library, and the assurance of free lifetime tech support. For the AmigaDOS edition, Comeau C++ currently supports SAS/C as its back-end code generator, but plans to add Manx C and Dillon/DICE C support soon. Under Amiga Unix, the language works with GNU C in addition to the provided AT&T compiler.

To learn more, write or call Comeau Computing at 91-34 120th St., Richmond Hill, NY 11418, 718/945-0009, or contact them on BIX (comeau), CompuServe (72331,3421), or Usenet (attmail.com!csanta!c++).

Remote Power

Sporting 24 input/output control ports, the **RCU-200 Industrial I/O Control Unit** was designed for applications that require monitoring of many on/off-type inputs or operation of many on/off outputs. All inputs are filtered and diode protected, while outputs have 50V/.6 Amp drivers that are diode protected to switch inductive loads. You can plug the board into and control it from any RS-232/RS-485 serial port, or you can custom program it to be an independent controller. Enclosed within a 13.5×10×7-inch plastic case, the unit sells for $995. The board alone (model CCB-6) costs $695. To order or inquire, contact Advanced Control Systems Corporation, Old Mine Rock Way, Hingham, MA 02043, 617/740-0223.



The RCU-200 Industrial I/O Control Unit from Advanced Control Systems.

Wanted!

Tell us about your new products and developments, and we'll pass word along. Send your press releases to *The Amiga-World Tech Journal*, 80 Elm St., Peterbrough, NH 03458 or llaflamme on BIX. ■

Modular Programming in C

Divide and conquer your programming problems by creating isolated chunks.

By Doug Walker

ALTHOUGH IT SOUNDS frighteningly like something a prefab house broker might endorse, modular design can be a programmer's best friend. It lets you give your programs the greatest possible flexibility and can actually save you time in the long run. The modular approach splits the programming task into sections (modules) that perform a set of services for the other parts of the program. These modules then use function calls and special data structures to communicate.

While modern languages such as C++, Ada, and Modula-2 actually encourage modular programming, there is no reason why you cannot write modular code using the good old Amiga standby, C. Let's look at some principles of modular programming from the C programmer's point of view.

BREAKING UP IS GOOD TO DO

It is more trouble to design and implement a modular interface than to just hack out code, but modular programming offers great benefits in the long run. If the code for a task is isolated in a module, you can easily change it to work with other hardware or software. You can improve its efficiency by rewriting it in assembly or by changing its basic algorithm, without revamping the code outside the module.

Using a modular design also allows you to generate prototype applications more easily. I am a proponent of getting something to work—as a sanity check on my code—even if that something is crippled. You can write small, "stubbed out" versions of the functions in your module and test the rest of the program with them. Later, if a bug shows up in the module, you can write a stand-alone test program that calls its various functions to reproduce and locate the bug.

When you design a module, you are in effect isolating its contents from the rest of the program. This technique of "information hiding" is very powerful. For example, if your module's task is to read or write disk files, you can change the disk-file format simply by rewriting the module: the rest of your program will know nothing about the disk format, and so cannot possibly be affected. With proper coding, you can even support more than one implementation of the disk I/O module, meaning that you can read more than one kind of disk file without changing your program.

Far from the smallest benefit of modularity is the fact that you may be able to reuse modules in other projects. Why rewrite screen I/O or disk I/O modules when you can lift them virtually intact from another project?

EACH MODULE IS AN ISLAND

The "interface" of your module consists of the functions that users are allowed to call and the data structures they can pass back and forth. A key point is that your module must not use any global data defined by the rest of the program, and vice versa. The complete, total and final description of the interface should consist of the function prototypes, any data structures passed as parameters to those functions, and any #define values used as parameters or used with the structure elements. If your module shares global data with the rest of the program, it is not a stand-alone, isolated piece of code. Preserving this isolation is the single most important challenge you face when writing modular code.

How do you go about defining a module interface? First, you decide exactly what your module is going to do. Look at the overall problem and identify a set of related tasks that you can easily split from the rest of the program. You can, for instance, group together all disk I/O routines. Other candidates for succession are screen I/O tasks, graphics functions, and memory-management routines.

Once you have settled on functionality, you must determine which services the module will provide to the rest of the program. Most modules should call an initialization function before calling any others, and should run a termination function after all access to the module is complete. Depending on the module's complexity, it might need functions that can retrieve and set options to control the module's behavior. And, of course, every module must include functions that actually perform the module's main task.

Finally, you must decide on a set of data structures to communicate with the world outside the module. Note that I said "to communicate with." The data structures that the module uses internally may or may not correspond with these communication data structures; the primary function of the latter is to provide a framework for the module and the outside world to pass information to each other.

FOR EXAMPLE...

I recently started a project with a friend of mine, Jere Yost, to implement a "game master's assistant" for a fantasy role-playing adventure. Jere and I wanted our program to list what each character in the game is currently doing, allow that list to be edited, and alert the game master when the task is complete so he or she does not lose track of what is going on. Unfortunately, Jere doesn't have an Amiga—he has an IBM PS/2—so the system needed to be able to run under MS-DOS as well as AmigaDOS.

Rather than put #ifdef statements throughout the code to deal with Turbo C screen I/O versus Intuition screens, we designed an interface using a generic set of screen-manipulation routines, then implemented that interface on both machines. ►

We called the interface the GMI (Game Master Interface) because I have a fondness for TLAs (Three-Letter Acronyms) and we needed a short prefix that we could place in front of the various routine names to identify them as parts of the module.

In this case, the entire interface is defined in the include file gmi.h, which is included by both the caller and the GMI implementation on each machine. The important pieces of gmi.h are reproduced in Listing 1; the complete file is on the accompanying disk in the Walker drawer. Any private information the Amiga implementation requires is placed in the file amgmi.h; private information for the PC implementation is in pcgmi.h. The plan was for a C source file that is above the GMI layer to compile unchanged on both machines, and with no #ifdef's.

The first things that went into this interface were the initialization and termination routines, GMIinit() and GMIterm(). In the case of the Amiga GMI, GMIinit() opens a screen and a window, and allocates some memory. GMIterm() closes the screen and window and frees the memory.

Because the module needs to handle multiple windows eventually, I chose to implement it with absolutely no global data; instead, the initialization routine allocates a data structure and returns it to the caller as the "handle" parameter. The caller does not know what it points to, but he faithfully passes it back to me as the first parameter of each function call in the interface, as you can see. If the caller wants more than one window, he can call the GMIinit() routine again—repeatedly, if necessary. I do not depend on global data, but I can very easily convert the code in the Amiga GMI implementation into a shared library, which allows multiple tasks to use the GMI code simultaneously.

Notice that the handle parameter in all the above calls is a typedef that points to a struct GMIHANDLE. This structure is not defined in gmi.h, because I did not want the higher layer knowing where the handle points. The definition for struct GMIHANDLE is actually in either amgmi.h (for the Amiga side) or pcgmi.h (for the PC side). Neither of these files is included by the caller, so there is no way for him to mess around with my private data if he does not have the structure definition. ANSI C allows you to use pointers to undeclared structures like this, which is very helpful. By using an undefined structure tag, you still get meaningful warning messages when you pass the wrong parameter type, but preserve the ability to hide the contents of the structure from the caller.

NO COMPUNCTION REGARDING FUNCTION

After the init and term routines, we determined what functions we would need. We decided to defer any bitmapped graphics support for a while and stick with text I/O to im-

Listing 1: GMI (Game Master Interface) for Portable Screen I/O

```
typedef struct GMIHANDLE *GMIHandle;

struct GMILOC { short x, y; };

struct GMIKS {short key, mod; };

struct GMIEVENT
{
   short type;              /* Event type; see defines in gmi.h */
   struct GMILOC loc;       /* Location of event */
   GMIHandle handle;        /* Window event occurred in */
   union
   {
      struct GMIKS k;       /* Keystroke description structure */
      char *data;           /* Additional data based on event type */
   } u;
};

int GMIinit(GMIHandle *handle);      /* Initialize GMI module */

int GMIterm(GMIHandle handle);       /* Terminate GMI module */

int GMIclear(GMIHandle handle);      /* Clear screen */

int GMIceol(GMIHandle handle);       /* Clear to end of line */

int GMImoveto(GMIHandle handle, struct GMILOC *loc);
 /* Move to (x,y) */

int GMImode(GMIHandle handle, int mode);
 /* Set text mode (see defines) */

int GMIwhere(GMIHandle handle, struct GMILOC *loc);
 /* Query current position */

int GMIinput(GMIHandle handle, struct GMIEVENT **event);
 /* Get input event */

int GMIputs(GMIHandle handle, char *string);
 /* Put string at current loc */

int GMIbox(GMIHandle handle, struct GMILOC *loc, int mode);
 /* Draw box to loc */


/* Define values for type field of GMIEVENT structure */
#define GMIE_NONE  0      /* Null value, internal use only */
#define GMIE_TEXT  1      /* User entered text in a field */
#define GMIE_KEY  2
 /* User entered a keystroke in a non-field */
#define GMIE_QUIT  3      /* End program*/
#define GMIE_MOVE  4
 /* Use changed cursor position (mouseclick, arrow hit) */

/* Define values for mode parm to GMImode and GMIline */
#define GMIM_NORMAL  0
 /* Normal text ; boxes overwrite old values*/
#define GMIM_HIGHLIGHT 1
 /* Highlighted text ; boxes XOR old values */
#define GMIM_LABEL  2
 /* Label text ; meaningless for boxes */

/* Define number of rows and columns on the display (1-origin!) */
#define GMI_ROWS 50
#define GMI_COLS 80
```

plement the system simply. We wanted the final product to be similar to a spreadsheet, with each bit of information on events taking up one line, and the columns containing such information as who is performing the action, what the action is, who is the recipient of the action, and so forth.

After some thought and a little browsing in the Turbo C manual, we determined we would need the following functionality to implement the target program:

- Clear the screen—GMIclear()
- Clear a line—GMIceol()
- Move to a specified cursor position—GMImoveto()
- Display text at least three different ways: normal, highlighted for important information, and another kind of highlighting for column labels—GMImode(), GMIputs()
- Determine the current cursor position—GMIwhere()
- Get an "input event" from the user (input events are defined to be any meaningful action that might require us to do something, like hitting a key on the keyboard or clicking with the mouse)—GMIinput()
- Draw a rectangle on the screen in one of two different ways: either destroy the old contents or overlay the old contents—GMIbox()

The portable layer would use the various GMI calls to display data on the screen, then call GMIinput() to get a keystroke or mouse event telling it what to do next.

Once we determined which functions were necessary, we had to define their parameters. To represent a cursor position, we chose the GMILOC structure, defined in Listing 1. GMImoveto(), GMIwhere(), and GMIbox() take a pointer to a GMILOC structure.

We wanted the GMImode() function, which sets the current text mode to normal, label, or highlight, to be able to take a parameter that specifies which mode to change to. We defined this as an "int" and #defined it to take the values GMI_NORMAL, GMI_LABEL and GMI_HIGHLIGHT.

GMIputs(), which prints a string to the screen, needs only to take the actual string, as the GMImode() call sets the text mode previously and the GMImoveto() call sets the position.

The most complicated function is GMIinput(). GMIinput() has to be able to describe what the user has done: Has the user pressed a function key, typed a character, clicked with the mouse, or taken some other action? We defined the GMIEVENT structure to describe input events. That structure contains a GMILOC structure that tells the position where the event took place and a handle field that identifies the window where it took place (for now, it is always the same as the handle passed in to GMIinput()).

It also contains a structure of type GMIKS, with two members: key and mod. If the user types a normal keystroke, key reveals the character typed. If the user selects a function key or clicks the mouse, key displays a special code (also defined in gmi.h), and the mod field tells us whether the user is also holding down either SHIFT key, either ALT key, the CONTROL key, or one of the two Amiga keys. We figured that this would allow us to define special meanings for about 40 or 50 likely key combinations, many more than is actually necessary.

Not every input event is a keystroke, and GMI can allow for menus as well. To accommodate them, we put a type field in the GMIEVENT structure that a program can set to various values according to whether the input event is a keystroke or something else. To determine which menu item has been selected, we need additional data, so we made the GMIKS structure part of a union that also includes a char * pointer. As new event types become necessary, we can easily add members to the union without changing the code that deals with the keystrokes.

CELL STRUCTURE

With all this defined, we were almost ready to go. First, however, we had to define a coordinate system to address the screen. Because we had a spreadsheet-style interface, we decided to use character cells. Thus, the GMILOC structure's x and y fields would contain the column and row of the character we needed. Furthermore, we chose to base the coordinate system on a 1-origin rather than 0-origin to jive with the Turbo C routines. This means that (1,1) is the upper-left corner of the screen. The lower-right corner is determined by a pair of #defines in gmi.h to be (80,50), because the maximum size possible on the PS/2 with the default font is 80×50 characters.

That done, Jere and I split up to implement our respective GMIs. I finished the Amiga GMI in about six hours. Part of that time I spent implementing a very thorough test program that uses the various parts of the interface. This test program has nothing to do with the final game-master-assistant program, but it can use the GMI interface quite easily. This is one of the great benefits of modular design—you can test modules with stand-alone test programs rather than discovering bugs and design flaws after the whole package has been written and integrated.

Once I had Amiga GMI working with the GMITest program, I got together with Jere and we ported GMITest to his machine. It nearly worked, but there was one slight problem: The Turbo C routines he was using (from conio.h) did not support 80x50 text! The GMITest program almost worked again when we changed the dimensions to 80×25, but we did not think that was good enough for the final program.

Jere decided to totally redo the PC GMI interface using graphics calls instead of console I/O calls. Again, we got together in a few days, and this time we had full 80×50. The GMITest program compiled using SAS/C on the Amiga, and successfully put up several screens of data. The same C source code, with no #ifdefs, compiled and ran on the PS/2 with the same result. Not only that, but the PC GMI implementation changed radically without affecting the GMITest source code at all!

While my friend was revamping his GMI, I started on the actual program. I had gotten quite far into it by the time he finished, and once we got GMITest working, I ported the code I had written. We ran into a couple of minor problems having to do with the fact that int variables are two bytes in Turbo C, rather than four bytes by default in SAS/C. I changed some data types and #define constants, and recompiled everything on the Amiga. It still worked fine, so we ported it again to the PS/2. This time, everything compiled and ran perfectly. I think we spent more time figuring out how to transfer the files from the Amiga to the PS/2 than we spent working on the actual code—the porting process took just one evening.

AS TIME GOES BY

GMI as it stands is fairly primitive. I would like to see it open a window on the Workbench screen instead of on its ►

own screen. I'd also like to be able to resize its window, and to see menus, scroll bars, and more complete mouse support. It's neat that all these things are easy to implement, and I can add them to the program whenever I choose. Further, almost none of my improvements will affect my friend's implementation on the PS/2.

Let's say, for example, that I open a window on the Workbench screen and give it a sizing gadget and a scroll bar. GMI defines the window size to be 80×50, but nothing says I cannot open a smaller window and use the scroll bar to get to the bottom half of that 80×50 space. The current GMI implementation does open its own screen, so it knows it can get 80×50, but an implementation that allowed resizable windows could easily open on the Workbench screen. If I had a production program using GMI, I could drop in this new feature and—presto!—that program would run on the Workbench in a resizable window. The PS/2 version would never need to know of the change.

The C language is very powerful and flexible. Because it is, you can write some very, very bad code—code that a more structured language like Modula-2 or Ada would not allow. Restrictions that force clean, modular code are not built into the language.

You, the programmer, must impose restrictions on yourself. Abide by them religiously. If you do, your code will be smaller, thanks to the absence of duplicate code and unnecessary checks for error conditions. It might also be faster, because you can isolate the time-critical portions and rewrite them at will. Above all, your code will be much more maintainable and extensible. ■

Doug Walker is a founding member of The Software Distillery and currently works for SAS Institute Inc. on the SAS Development System for AmigaDOS. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (djwalker) or USENET (walker@unx.sas.com).*

Five Steps for Writing a Good Module

1. Decide what the module will do.
2. Choose a function that will do what you need.
3. Define the needed data structures, constants, and macros as parameters to the functions.
4. Write the code for the functions.
5. Write a test program to exercise the module.

The Peer Review Board Philosophy

When compared to other contemporary platforms, the Amiga is incredibly complex with an often overwhelming number ROM-resident and disk-based routines. Yet, these same routines let you easily create programs in a minimal amount of code. The trick to mastering the machine is, of course, study and practice. To help you, *The AmigaWorld Tech Journal* will provide a continuing supply of techniques and examples that not only work, but also operate by the rules. Proper programming practices are an absolute necessity in a multitasking environment.

The problem is that very few individuals are experts in every aspect of the Amiga and are fluent in every computer language. Therefore, no one person can check the accuarcy of every article we publish. To maintain the high degree of credibility we (and you) demand from the *Tech Journal* we instituted the Peer Review Board.

Before an article appears in print it is examined and critiqued by at least two Peer Reviewers who are recognized by the Amiga technical community as experts in the subject. Based on their comments we publish the article, have it rewritten, or reject it.

Over the years, the members of the Peer Review Board have demonstrated their technical expertise by developing respected products for the Amiga. Many members played critical roles in the initial development of and continuing improvements to the Amiga's hardware and operating system. Rest assured with these watchdogs on the job, the articles you read will teach the latest techniques and standards in the best manner possible. The Peer Reviewers' extra efforts make your job much easier.

The Peer Reviewers

Rhett Anderson
Gary Bonham
Gene Brawn
Brad Carvey
Joanne Dow
Keith Doyle
John Foust
Andy Finkel
Eric Giguere
Jim Goodnow
Scott Hood
David Joiner
Sheldon Leemon
Dale Luck
R.J. Mical
Eugene Mortimore
Bryce Nesbitt
Carolyn Scheppner
Leo Schwab
Tony Scott
Mike Sinz
Richard Stockton
Steve Tibbett
John Toebes
Mike Weiblen
Dan Weiss
Ben Williams

From p. 27

from the procedures involved with producing graphics input/output.

CHANGE THE WORLD

This modest introduction to geographic mapping shows how you can use basic rotation matrices to simplify world mapping and where to find some map databases. This should be enough to let you embark upon your own geographic mapping project. There are many aspects of world mapping yet to be addressed, such as equations for the Van der Grinten projection and a way to simplify handling the eccentricity of the earth. The future shape of the world is now in your hands. ■

Howard C. Anderson is a member of the technical staff of the Technology Computer Aided Design group of Motorola's Advanced Technology Center, which develops new processes for manufacturing semiconductor chips. He also is the author of the GWIN graphics system (found on Fred Fish disk #433) and sings tenor in a bluegrass band called BluegrAz Express. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

References

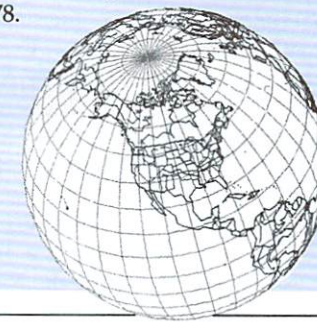
1. Johnston, William D. "Computer Generated Maps," *BYTE*, May and June 1979.

2. Miller, Robert, and Francis Reddy. "Mapping the World in Pascal," *BYTE*, December 1987.

3. Foley, J. D., and A. Van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, 1982.

4. Hearn, Donald and M. Pauline Baker. *Computer Graphics*. Prentice-Hall, 1986.

5. Maloney, Elbert S. Dutton's *Navigation & Piloting*. Naval Institute Press, 1978.



Building an ARexx Function Host

By Eric Giguere

IF AREXX DOESN'T offer a capability you want, you can simply write a function or set of functions to implement that capability. You can write the function in ARexx itself—an external function—or in another language as part of a function host or a function library. External functions are just short ARexx programs that other ARexx programs can call as functions. Their use is well-documented, as are those of function libraries. Function hosts, however, are not as well understood. This article explains what a function host is and shows you how to build one.

The RexxPath function host (with full source included in the accompanying disk's Giguere drawer) will serve as one of our examples. To compile the RexxPath program, you'll need either SAS/C 5.10 or Manx Aztec C 5.0d. If you aren't using the Workbench 2.0 header files, make sure you install the ARexx header files from your ARexx disk. You also need to be familiar with interprocess communication—see "Pass the Word: Interprocess Communication" (p.35, June/July '91).

REXXPATH

ARexx expects to find files in either the current directory or the REXX: directory, if no absolute path is specified when a program or external function is invoked. As a consequence, the REXX: directory tends to fill quite quickly with your own programs. There's no easy way to group related files into separate directories for ARexx to search. Actually, that's a lie. You could use a program like PathMan (included with Bill Hawes' WShell) or PathAss (from the Fish disks) to assign the logical REXX: directory across several physical directories. But why spoil our fun?

RexxPath allows you to add a list of directories you want searched whenever an ARexx program calls a function that cannot be found in the current directory or the REXX: directory. It also allows you to specify aliases for external functions. Unfortunately, RexxPath does have limitations. Because it's a function host, only function calls are intercepted. If an application tries to start an ARexx program directly, that program will be sought using the normal rules only.

You start RexxPath using an ARexx script, rxpload.rexx. The script takes a filename as its only argument, such as:

```
rx rxpload sample.init
```

Note that RexxPath must be in your path for the script to work. The script starts RexxPath and then reads your input file to add the paths and aliases you want RexxPath to use. The input file format is simple:

```
[paths]
ram:arexx
hdisk:sources/arexx

[aliases]
ParseFilePath PF
MyDate   fh1:special/date
```

The first section is a list of paths to search when a function call is made. The second half is a list of alias/filename pairs. If the MyDate function is called, for example, RexxPath will actually call the fh1:special/data.rexx file instead. (If no absolute path is given for the alias, RexxPath will search its path list as well.) RexxPath is not case-sensitive—the "mydate" and "MyDate" functions are equivalent.

If you look closely at the rxpload script, you will notice that all it does is parse the input file and send a stream of commands to RexxPath. In other words, the RexxPath program is also a command host. The sample initialization file above could have been easily written as an ARexx program:

```
/* Initialize RexxPath */

address RexxPath
'clear paths'
'add path ram:arexx'
'add path hdisk:sources/arexx'
'clear aliases'
'add alias ParseFilePath PF'
'add alias MyDate fh1:special/data'
```

In addition to these commands, RexxPath also supports commands for retrieving the current path and alias lists:

```
/* Show RexxPath state */

option results
address RexxPath
'get paths ;' /* use ; as separator */
say "Current search paths:" result
'get aliases'
say "Current aliases:" result
```

RexxPath can be halted at any time by sending it a 'quit' command:

```
rx "address rexxpath quit"
```

or by using the AmigaDOS BREAK command.

DEFINITIONS AND LISTS

Before we can talk about how RexxPath works, you need to understand a few terms. Running in the background on

Follow the RexxPath example to add more flexibility to your programs.

your Amiga is the ARexx resident process, the program that's started by the rexxmast command. The resident process is responsible for starting ARexx programs and for keeping track of special resources, such as the library list. The library list is the list of function libraries and function hosts that the resident process knows about. When an ARexx program calls an external function (a function that is not defined in the ARexx program and is not built into the ARexx interpreter), the program searches down the library list, asking each library or host if it supports the function. The search stops with the first library or host that acknowledges the function, otherwise a "function not found" error is returned to the ARexx program.

A function library is a shared library that knows how to respond to function requests. For more on libraries, see "Extending ARexx," p. 18.

A function host is just an application program that knows how to process function requests. In fact, any application that supports ARexx commands (a command host) can be easily modified to support function calls as well. The resident process is a function host, installed at a low priority in the library list. When it gets a function request, it searches the current directory and the REXX: directory for a filename that matches the function name. Usually no other function libraries or hosts are in the library list after the resident process, making the file search the last step in the search for an external function. RexxPath installs itself after the resident process to intercept those function calls that the resident process cannot handle.

AREXX PORTS AND MESSAGES

ARexx relies heavily on Exec's interprocess communication facilities. ARexx programs are started by sending a message to the resident process. ARexx programs send commands to applications with messages. As you might expect, then, function requests are sent to a function host via messages as well.

To receive messages, a function host must first create an ARexx port, a named message port. This port is then registered with ARexx by using the ADDLIB() built-in function:

```
call addlib "REXXPATH", -61
```

The two parameters to ADDLIB() are the name of the port (case is important) and the priority at which to insert it into the list. Because the resident process' port ("REXX") has a priority of –60, we install the "REXXPATH" port just below it. (Note that a C program can send a special message to the resident process to add an entry to the library list instead of relying on ARexx scripts to do it themselves.)

Messages received at an ARexx port are expected to be in a special format known as a RexxMsg, which defines the protocol that all ARexx-aware programs should follow. The RexxMsg structure is defined in the rexx/storage.h header file:

```
struct RexxMsg {
    struct Message rm_Node;
    APTR           rm_TaskBlock;
    APTR           rm_LibBase;
    LONG           rm_Action;
    LONG           rm_Result1;
    LONG           rm_Result2;
    STRPTR         rm_Args[16];
    struct MsgPort *rm_PassPort;
    STRPTR         rm_CommAddr;
    STRPTR         rm_FileExt;
    LONG           rm_Stdin;
    LONG           rm_Stdout;
    LONG           rm_avail;
};
```

As you can see, it's an extension of the Exec Message structure and thus can be used with the usual PutMsg(), GetMsg(), and ReplyMsg() functions. The RexxMsg structure is used in all communications between ARexx programs, the resident process, and application programs that support ARexx. So in fact, an ARexx port can also be used to receive command messages, not only function calls.

Most of the fields in the RexxMsg structure are of no interest to us. The first three are used by ARexx when sending a command message and should be considered very private. The last six are defined only by an application sending a command message to the resident process.

GETTING THE FUNCTION CALL

Processing a function call is a three-step process: Receive the message and check its validity. Call the appropriate routine to execute the function. Finally, set the result codes and reply to the message.

When a message arrives at your host's ARexx port, the host should first make sure it is a function call, because the same message structure is used for all ARexx communications. A function call will have the RXFUNC flag set in the rm_Action field of the RexxMsg structure:

```
struct RexxMsg *msg;
msg = (struct RexxMsg *) GetMsg( port );
if( ( msg->rm_Action & RXCODEMASK ) == RXFUNC ){
    /* process function */
}
```

RXCODEMASK and RXFUNC are constants defined in rexx/storage.h. If the RXFUNC flag is not set, then the message is either a command message (the RXCOMM flag is set instead) or an unknown message.

An RXFUNC message will have the name of the function being called stored in the rm_Args[0] field. You can treat this as a normal C-style string pointer:

```
if( stricmp( msg->rm_Args[0], "myfunc" ) == 0 ){
   /* code for myfunc() */
}
```

Your function host must keep a table of all the function names it supports and the actual C routines that implement the functions. When a function call comes in, the host searches through this table to call the correct function. (The name search is usually case-insensitive.) Make the search as quick as possible, because, if a host does not support the function ARexx is asking for, it should reply to the function call to let ARexx continue the search with the next host or library.

A function call will probably have at least one argument string. These arguments will be stored as string pointers in the rm_Args[1] to rm_Args[15] fields of the RexxMsg structure. (Note the 15-argument limit.) The actual number of argument strings can be extracted from the rm_Action field as follows:

```
nargs = ( msg->rm_Action & RXARGMASK );
```

Some of the arguments may have been omitted, in which case the corresponding rm_Args entry will be NULL. In this case you should supply a suitable default value.

The actual implementation of a function is up to you. Just remember that the arguments you get from ARexx are all strings and you must perform any necessary conversions.

RETURNING RESULTS

By definition, an ARexx function always returns a result string if no error occurs during the function call. So, before you can reply to the function call you must set two values—a return code and a result string.

The return code is an integer that tells ARexx whether a function call succeeded or not. It should be 0 if the call was successful and nonzero (10 is the usual value) otherwise:

```
msg->rm_Result1 = 0; /* call succeeded */
```

If the return code is 0, ARexx expects a result string in the rm_Result2 field. You set a result string by calling the ARexx system library function CreateArgstring():

```
msg->rm_Result2 = (LONG) CreateArgstring( "573", 4 );
```

The first argument to CreateArgstring() is the string to be created, the second argument is the string's length plus one. ARexx will free the result string when it is no longer needed. (Calling details for CreateArgstring can be found on the disk.)

What if an error occurs? If rm_Result1 is nonzero, no result string should be sent. Instead, store an integer error code in the rm_Result2 field.

If your function host doesn't support a function ARexx is searching for, you should set the result fields as follows:

```
msg->rm_Result1 = 5;
msg->rm_Result2 = 1;
```

ARexx will treat this error sequence as a special case and continue the function search with the next host or library.

Once the result fields have been set, you can ReplyMsg() the RexxMsg structure to return it to the ARexx program that called your function host.

COMMAND PROCESSING

A function host can easily be modified to be a command host as well. Commands are simply strings that ARexx evaluates and sends to an application's ARexx port. They are sent using the same RexxMsg structure described earlier, but the rm_Action field has the RXCOMM flag set instead of RXFUNC, and the complete string is found in the rm_Args[0] field. The other 15 slots are all empty. When a command arrives, the application must parse the command string. Return codes and result strings are also used with commands, though a result string is only sent when the RXFF_RESULT flag is set in the rm_Action field. (The RXFF_RESULT flag will be set if the OPTION RESULTS instruction is used in the ARexx program that sent the command.)

FUNCTION HOSTS ON DISK

You will find two function host examples in the accompanying disk's Giguere drawer. The first example is, of course, RexxPath. The second is a small function host called RGB, which lets you play with the Workbench screen colors within an ARexx program. Both function hosts use SimpleRexx, a set of C routines for implementing ARexx interfaces. SimpleRexx was originally developed by Mike Sinz, and I added some additional functionality and compiler support. Feel free to use SimpleRexx in your own applications. You will find full documentation for SimpleRexx in the *Amiga Programmer's Guide to ARexx*.

Some implementation details: RexxPath keeps a list of search paths and function aliases. When a function call arrives at the REXXPATH port, RexxPath searches the list for an external file that matches the function call. If a file is found, the full pathname of the file and the function arguments are sent as a new RXFUNC message to the REXX port, which will then load and execute that file. When the file has finished executing, a reply message will arrive at the REXXPATH port. RexxPath will then copy the results of that message over to the original function call message and reply to it. In effect, RexxPath acts as a message relay. RexxPath also accepts command messages to add and clear paths and aliases.

RGB is more of a conventional function host than RexxPath. It supports four functions: SetRGB() to set a color register's RGB values, GetRGB() to get a register's RGB values, ResetColours() to reset those values, and QuitRGB() to exit the function host. Details on the use of these functions are found on the disk in a sample ARexx script.

RexxPath isn't perfect, but it shows the flexibility of function hosts. This article has just scratched the surface—details on interfacing to ARexx could (and do) fill a book. Study the listings and text files on the disk carefully before you build your own function host, but don't hesitate to experiment. If you do create a useful host, by all means pass it on to your friends. We'll all thank you! ■

Eric Giguere is the author of the forthcoming Amiga Programmer's Guide to ARexx *and a member of the Computer Systems Group at the University of Waterloo (Ontario). Write to him c/o* The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or contact him on BIX (giguere) or Internet (giguere@csg.uwaterloo.ca).*

Designing the User Interface: A Matter of Principle

Learn the basics of user-interface design in this first-of-a-series article.

By David "Talin" Joiner

MOST SUCCESSFUL SOFTWARE creators today realize the importance of a well-designed user interface. Unfortunately, many of them do not have a clear idea of what "well-designed" means. Often they know only what *not* to do, having learned from painful experience.

One important tool in designing a user interface is a standard. A good standard can take a lot of the guess-work out of interface design, as well as provide users with a sense of familiarity.

Standards, however, are limited in two ways. First, some types of applications are so unusual they exceed the scope of a standard. Second, standards are general and don't specify the many fine-grained details of an individual application; those must be determined by the software's designer.

Commodore has recently compiled a new book, *Amiga User Interface Style Guide* (published by Addison-Wesley Publishing Company, ISBN 0-201-57757-7), that will have a major impact on future Amiga applications. The guidelines it puts forth are not only a standard, they're a good standard. From the designer's point of view, user-interface design is subjective, and each designer has personal opinions. Yet, to my surprise, I find little to disagree with in this book.

This article is the first in a series that will explain the elements of good user-interface design, covering not only much of the material in the new style guide, but various topics that go well beyond. We'll examine ideas taken from other style guides, from academic discourses on user interfaces, and from the practical experience of professional software developers. While we'll start off with general principles, subsequent articles will be more specific and contain real-world techniques and examples. (If you bought the *Amiga User Interface Style Guide*, you may notice a more than casual resemblance between some of my text and certain sections of the book. In a few cases this is because I'm paraphrasing ideas taken from the book; in many cases, however, it's because the writers of the style guide used material I provided.)

METAPHOR

Most graphic user interfaces today are designed around a "metaphor"—a set of behaviors based on a real-world object or mental construct that most users know. While it's not necessary for a well-designed interface to follow a metaphor, the metaphor does provide a solid collection of consistent, familiar behaviors to model as a shortcut in designing an interface that's also consistent and familiar.

Metaphors give you a conceptual handle on the side effects of program behavior. There are many possible side effects of each user action, in addition to the main effect. The metaphor can suggest which side effect is most natural and pleasing to the user.

PHYSICAL VERSUS CONCEPTUAL

In general, the interface designer wants to find some common physical system on which to base a metaphor. Physical systems have an advantage in their familiarity.

For example, for a music sequencing program one possible model would be a multitrack tape recorder. This is a good model, because it immediately suggests a number of familiar features that can be applied to the program: play, fast-forward and rewind buttons, a tape counter, VU meters for watching the sound level, and so on.

There are times, however, when you'll want to create a set of behaviors based on a completely abstract set of concepts. This is especially true when no relevant physical model exists. For example, a hypertext database system has no physical analog, and, while it's powerful and simple, it's also a completely unfamiliar concept to most people.

A 3-D solid-modeling application could be designed to act on wireframe objects as though they were either physical objects or abstract mathematical entities. In the former case, you would rotate the object by grabbing a corner and pulling it, just as you would grab the corner of a real object. You could squash in a similar fashion, by grabbing and pulling. (Note the difficulty, however, in the two actions being confusingly similar). In the case of an abstract model, you would use the mouse to adjust a rotation knob or a squash slider. This scheme has the advantage that conceptually similar user controls can be grouped together—the knobs for rotation on the X, Y, and Z axes can all be next to each other, while separate from the X, Y, and Z scaling controls.

OBJECT-ACTION

Most of the popular graphic user interfaces today use the "object-action" method, also known as the "noun-verb" method. This simply means that first you select the thing you want something done to, then you select what you want to do to it. For example, you click on the icon or object to be deleted and then activate a delete function, rather than choosing delete first and then clicking on the item.

The advantage of the object-action method is that commands tend to be highly localized and change little based on the previous command. Therefore, the user has little to remember in terms of interlocking sets of functions.

MODALITY

The opposite of the object-action method is the "action-ob- ►

ject" method, more commonly known as "modality." Many older programs use this "mode" concept, which involves a global change in the behavior of the program. For example, some older text editors had a text-insert mode and a command mode, and the editor would behave completely differently in each. This confused many people, because they couldn't remember which mode the program was in. In many cases, modality was so abused (five or more different modes looking exactly the same but acting completely differently) that now modes are considered bad by many user-interface experts.

Sometimes, however, modes are necessary. A paint program with a tool bar is a good example of a modal program where the user tends to perform the same action many times in a row. (Note that real brushes and paint are highly modal, so the paint program is following a physical model.)

How does the paint program avoid confusing the user? The answer is simple but also demonstrates a fundamental user-interface rule, one that I call "visual parallelism."

"Anytime an application changes its behavior, it should exhibit a corresponding visual change."

Rule: Anytime an application changes its behavior, it should exhibit a corresponding visual change. The size of the visual change should be proportional to the size of the behavior change.

Corollary: Any major visual change (other than in a drawing or editing area) should be in response to a behavior change.

More generally, an application's exterior should reflect the state of its interior. I added the corollary, by the way, because I've noticed a number of applications that seem to play with their display for no good reason.

FOCUS

When you look at a computer screen, your attention probably concentrates on one particular spot. Therefore, most applications limit activity to a small area of the screen. To draw your attention, that area may be animated, because we humans have a lot of motion-detecting hardware in our eyes. Examples are the text cursor in a word processor and the currently selected line or polygon in a CAD program, all of which blink. Even when nothing else is happening, the mouse pointer itself is an animated object that is likely to be the focus of the user's attention.

It's important to identify the user's focus to use as a communication channel. You can animate or change the appearance of the focus object to indicate some change in the program's status. The Busy (or sleeping) pointer is an example of this. If the Busy message appeared in the title bar or on the currently selected icon, instead of as the mouse pointer, it would be much less effective.

Similarly, many word processors indicate a change in behavior by changing the text cursor. In one text editor, while you're saving a file to disk, the cursor changes to half its normal height, indicating that you can move around and look at different parts of the file, but you can't make any changes. You're sure to notice that something has happened, even though 99.9% of the pixels on the screen haven't changed.

Another focus for the user can be the object that's just been operated on. For example, when an application is launched from the Workbench, until that application's window opens the user's attention will probably be directed toward the program icon.

FEEDBACK

Users feel much more confident when each of their actions produces an immediate visual reaction. This is true even if the action is computationally intense—at the very least users expect to see some indication from the program that "I'm working on it." Without feedback, a user may get the false impression that the machine is broken.

I like to provide as much feedback as possible, and through as many sensory channels as possible. For example, when a user drags a note in a music program, he should not only see the note move on the screen, but hear it change pitch. After all, the visual graphic of the note is only a representation of what the user is really interested in, the sound of the note.

In general, it's easy to provide instantaneous feedback for quick actions. For example, when you type a character in a text editor, the character appears immediately. For tasks that take longer, some kind of surrogate feedback is necessary. A familiar example is moving windows: You don't actually drag the window, you drag an outline of it. Similarly, when you move a complex object in a CAD program you often actually drag a simpler proxy object. This type of visual special effect is known as an "echo" in user-interface literature.

Another technique for achieving responsive feedback is "partial updates," in which only the part of the screen near the focus is updated rapidly, while other parts are updated more slowly. A word processor may refresh the line you're typing on each time you hit a character but redraw the other lines only when it has a few machine cycles to spare. The multitasking nature of Intuition events makes this quite easy, although not obvious at first glance. (I'll cover this and other techniques in a future article.) It's an interesting and challenging problem to make the user think the computer is faster than it really is—the illusion of responsiveness (which, from the user's point of view, is as good as the real thing).

CLUTTER VERSUS SIMPLICITY

Product developers feel constant pressure to cram more and more features into their applications. Unfortunately, many of these new features also require new controls, each of which must be properly set and adjusted or the program may not function as the user intends. Wendy Carlos, one of the pioneers of electronic music, has a saying that sums up this situation well: "Whatever you *can* control, you *must* control."

Having a lot of controls makes it difficult to lay out the display. Buttons must not be too small, icons must be readable, and proportional gadgets must be large enough to have some degree of fine control. In addition, the controls should be grouped logically, and the display must have a pleasing appearance.

OVERLOADING

One way to reduce screen clutter is to "overload"—that is, give objects more than one meaning, or function. A good ex-

ample is the circle-drawing gadget in Electronic Arts' Deluxe-Paint III: If you click on the upper-left portion of the gadget, it draws a hollow circle; the lower-right draws a filled circle. From a technical standpoint, these are really two different controls, but that's not how the user perceives it. He sees a single gadget with two different functions.

Overloading must be used with great care. The different functions assigned to one control must all fall within the same category in the user's mind. In addition, the control must be clearly marked as special; DPaint uses a diagonal line through the button (which turns out to be insufficient, but the alternatives are limited). Finally, an overloaded control should be limited to just a few meanings.

NOVELTY

Often an application will deviate from an accepted interface standard simply to attract more customers with its newness. Some home and hobby users want a change from the controls they're used to, and may even like to feel they're participating in the evolution of software by buying an application with a new user interface. In the same vein, users with an antipathy toward conformity may dislike boring, colorless, business-like programs. Witness the number of popular disk-copying programs with Copper-list animated backdrops or intense gadget artwork.

These users who value newness must be balanced with the large number of novice and serious users who prefer tight standardization. The key to this balance is knowing your market, both the customers of today and those of tomorrow. Is your typical customer a businessman who "just wants to get some work done," a hobbyist who's excited by hi-tech innovations, an artist who wants an aesthetically pleasing environment, or a novice who's intimidated by complexity? Most likely, he will be some combination of all these.

I believe that breaks with convention, if done well, can get a moderate amount of positive user feedback (and also can look good on the sell sheets if you have a clever marketing department). However, this works only if you follow some very stringent rules:

1. Change no more than a few conventions. If *everything* is different, sales will be miserable.
2. The changes can't be too far from convention.
3. The changes must be clearly superior in either aesthetics or function, although perhaps only for the application involved.
4. Never, never break with convention simply because you, the designer, think it's "neat." That's the mark of an amateurish designer (like I once was—I've written megabytes of code that was "neat" but turned out to be "bad").

Consider these guidelines in designing all aspects of your program's user interface. They'll let you evaluate the "goodness" of your interface before you build it, instead of after all the work is done and the program is out the door. ■

David "Talin" Joiner is the author of Faery Tale Adventure and Music-X, plus an artist, award-winning costume designer, and moderator of the user.interface topic of the Amiga.sw BIX conference. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (talin).*

List of Advertisers

AmigaWorld Tech Journal is a publication of International Data Group, the world's largest publisher of computer-related information. International Data Group publishes over 150 computer publications in 49 countries. Over 25 million people read International Data Group's publications each month.

Subscription problems or address changes: Call 1-800-343-0728 or write to *AmigaWorld Tech Journal*, 80 Elm Street, Peterborough, NH 03458. **Problems with advertisers:** Send a description of the problem and your current address to: 80 Elm Street, Peterborough, NH 03458, ATTN.: Margot L. Swanson, Customer Service Representative.

The Finer Points Of Pointers

By John Toebes

THE FIRST STEP in mastering C is learning how to make the language work for you. By understanding a few simple rules, you can easily tackle almost any suitable task using pointers in C. Due to the cryptic nature of C, however, many people have difficulty making pointers and designing programs that use them correctly.

As a guide to pointers in C, we will examine the following aspects: how pointers (and the objects they point to) are declared, how to deal with pointers to pointers or pointers to arrays, and how pointer declarations work in parameter lists. (See "Pointer Fundamentals" for a discussion of what a pointer is and how it is represented.)

DECLARATIONS IN C

Because pointers are frequently used in C, the designers of the language decided to employ a simple notation to reduce the amount of typing. What can be perplexing is that the character that indicates indirection—the asterisk (*)—is also used to show multiplication. To eliminate confusion, when you are using an asterisk as a pointer, do not put any space after it; when you are performing multiplication, place a space on both sides of the asterisk. As you become more practiced at C, you will find it easy to distinguish the two by their usage.

When you declare a pointer to an object, you begin by declaring the object itself and then adding an asterisk. For example, a pointer to a character starts out as:

```
char p;
```

and ends up as:

```
char *p;
```

Note that you can apply this method to create a pointer to a pointer to an object by simply placing another asterisk in front of the existing one:

```
char **p;
```

The above declares four bytes that point to the four-byte location containing a pointer to a single character. It does not allocate the space for the pointer to the character, nor the space for the character itself. These require separate declarations, resulting in a total of nine bytes before the declaration is useful.

There is no limit to the number of levels of indirection you may add, but practicality steps in at about three. It is difficult to imagine three levels of indirection (a pointer to a pointer to a pointer to an object), much less ensuring correct usage. If you find yourself adding more than two asterisks, take that as a clue that something is wrong.

DECLARING ARRAYS

Now let's look at array declarations. Arrays, too, are often used with pointers. Start with the simple type of array and then put the array notation of square brackets after the item:

```
char p;
```

becomes:

```
char p[50];
```

giving you an array of 50 bytes. A two-dimensional array becomes:

```
char p[50][10];
```

The fun starts and the confusion sets in when you combine pointers with arrays. Always remember that an array declaration has a lower priority than a pointer declaration. Hence:

```
char *p[50];
```

declares an array of 50 pointers to characters (using 200 bytes of storage) while:

```
char *(p[50]);
```

declares a pointer (using four bytes of storage) to an array of 50 characters. In fact, because C lets you go beyond the end of an array without penalty, the above is equivalent to:

```
char *(p[1]);
```

or even:

```
char *p;
```

GOING STRAIGHT WITH POINTERS TO POINTERS

When working with a pointer to a pointer, keep in mind that there still has to be a base object to point to. In fact, when it has a single base type that it is pointing to, a pointer is virtually an array. To illustrate, take a simple example of a pointer to a character string:

```
void main()
{
   char *p = "something interesting";
   printf("p=0x%08lx *p='%c' p=\"%s\"\n", p, *p, p);
}
```

If you compile and run this program, you will see:

```
p = 0x00203498 *p='s' p= "something interesting"
```

If you think C pointers are a curse, the following guide will turn them into a blessing.

Of the three printed ps, the first is the address of what p points to—the start of the string "something interesting" in memory. The *p dereferences the pointer to give you the first character of the string—s. It is interesting that the third p makes it appear that you passed an entire string when, in reality, all you passed was the address of the first character. By special convention in C, a string is an array of characters that has a 0 byte at the end. Deep in the bowels of printf, there is a code that logically does something like:

```
void fake_printstring(char *p)
{
    int i;
    for (i= 0; p[i] != '\0'; i++)
        putc(p[i]);
}
```

where p is a pointer to a single character in memory. What C does is let you use any pointer to an object as if it were an infinitely long array of that object. The creative programmer may even find that this subroutine also works:

```
char peek(int addr)
```

►

Pointer Fundamentals

SO, WHAT IS a pointer anyway? Technically, it is a reference to—or points to, as its name implies—another object. By itself, a pointer does not convey any real information; it is what a pointer points to that is of real importance.

In C, there are many different types of pointers. The most common C pointer is a pointer to a character (or character string), which is declared as:

```
char *p;
```

This tells the compiler to reserve four bytes of storage (on the 68000; the size varies for other machines) to hold a pointer to a character. Note that this does not declare a place to put whatever the pointer is pointing to—quite like having a phone number for someone who has not installed a telephone. It helps to keep this perspective in mind so as to avoid some of the common mistakes made by beginning C programmers.

In addition to a pointer to a character, you can have a pointer to an integer, to a floating point number, to a structure or array of data, and to another pointer. Using the phone number analogy again, a pointer to a pointer can be compared to dialing a number and getting the recording that gives you a different number to dial. This is called indirection, or in the case of a pointer gone astray, misdirection.

Each pointer has the same representation: four bytes that indicate the address in memory. What the bytes point to and how C deals with them, however, are completely different. For this reason we speak of the type of a pointer in terms of what it points to. You can't use a pointer to a structure as a pointer to a character string without doing some kind of conversion—in C this is called a cast. Using a converted pointer does nothing at all to the representation of the pointer (it is still the same four bytes pointing to the same location in memory), but it does tell the compiler that you are accessing a different type of object. Think of this as each phone number telling you what you will find when you call each number.

Of course in practice, it always helps to have a generic pointer that can point to anything. In the past, this was char *, but with the advent of the ANSI standard, it is now:

```
void *
```

(pointer to void). Actually, a void * pointer doesn't really point to anything, although it is technically capable of pointing to everything. The C compiler recognizes when you assign to and from a void * pointer, and you don't have to cast it. If you attempt to dereference the void * pointer without a cast, however, you will get an error from the compiler. □ —*JT*

"The concept of a pointer as an array allows us to perform a few tricks—actually, common practices in C."

```
{
   char *p = 0l;
   return(p[addr]);
}
```

Here, you initialize p to point to the start of memory—location 0—and then use the address value passed in to index that array. Because C does not do any bounds checking, even if you do not allocate any memory, it lets you obtain access to something that you have no right to get.

To illustrate where people start having problems with pointers as arrays, consider a slight change to the example above:

```
int peekw(int addr)
{
   int *pI = 0l;
   return(pI[addr]);
}
```

If you pass a 4 to the first (char peek) routine, you will get the character at the fifth *location* in memory: 0x00000004. However, when you pass a 4 to the second (peekw) routine, you get the integer at the fifth *word* in memory: location 0x00000010. This occurs because C automatically scales the index by the size of the object being pointed to.

The concept of a pointer as an array allows us to perform a few tricks—actually, common practices in C. The first example is that of allocating storage to hold strings. Many times you will see the code:

```
char *p;
p = malloc(strlen(input)+1);
strcpy(p, input);
```

What this does is allocate an array of characters and return a pointer to the start of that array. Though you might also see a declaration of the form:

```
char p[] = "I'm an array";
```

it is not the same because it declares an array of characters of which p is the start, but does not declare a four-byte pointer (only the storage to hold the string). In contrast:

```
char *p = "I'm an array";
```

declares the four-byte pointer in addition to the storage to hold the string.

WORKING WITH LARGE BLOCKS OF DATA

If life were made up only of strings, everything would be simple; however, programs have to consider dynamic allocation of large blocks of data. Take, for example, a database in which records have to be inserted, deleted, and sorted. You could use:

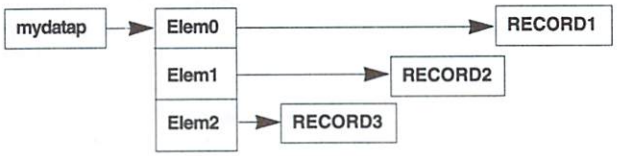
```
struct RECORD mydata[100];
```

to declare data for 100 records and then copy records each time they are moved. Insertion and deletion can require moving a lot of data, so you should consider a declaration such as:

```
struct RECORD **mydatap;
```

which declares a pointer to an array of pointers to your data. Although this takes up only four bytes, when you allocate the array and the individual data, you end up with something similar to:

Once you know the number of elements you have to deal with (which may come from a configuration option or from



scanning to see how much data there is), you can allocate the array with:

```
mydatap=
      (struct RECORD **)malloc(numelems*sizeof(struct RECORD *));
```

This allocates only the array; you still do not have anyplace for the data to actually go. Following good programming practice, you would use:

```
sizeof(struct RECORD *)
```

to obtain the number of bytes to allocate for each pointer (rather than blindly hardcoding a constant four for the size of the pointer). This also helps to serve as a reminder of what it is you are creating.

Once you have the array, you can allocate storage for the elements with:

```
mydatap[0] = (struct RECORD *)malloc(sizeof(struct RECORD));
```

or, in the case of the first element, you can use:

```
*mydatap = (struct RECORD *)malloc(sizeof(struct RECORD));
```

To reference the data from an element of the array, you can use:

```
mydatap[0]->field
```

Note that you cannot substitute:

```
mydatap[0].field
```

because you have two levels of indirection. You can identify this from the declaration for mydatap, which has two asterisks. In order to access the actual data, you need to do two indirections. Only right arrow (->), asterisk (*), and brackets ([]) imply indirection; a period (.) is simply a reference to a member of a structure.

Another trick for referencing this data is to manipulate the array by moving a pointer through it:

```
struct RECORD **this;
this = mydatap;
```

►

"Because C passes all parameters by value, it is not possible to modify any of the input parameters to return."

```
for (i = 0; i < numelem; i++)
{
   printf("Element %d has %d\n", i, *this->field);
   this++;
}
```

What you might attempt to do (with no success) would be:

```
struct RECORD *this;
this = *mydatap;
for (i = 0; i < numelem; i++)
{
   printf("Element %d has %d\n", i, this->field);
   this++;
}
```

While you did the right number of indirections to get to the data, this fails to work because you are treating mydatap as a pointer to an array of records instead of a pointer to an array of pointers to records. You can tell because this++ increments the pointer by the size of the record (remember that pointer increment is based on the size of the object pointed to, in this case a struct RECORD *). What is nice about this organization is that records can be swapped by simply exchanging the pointers. You can delete or insert a record by moving the pointers and leaving the data in the same place.

DECLARATIONS IN PARAMETER LISTS

Once you feel comfortable manipulating data with two levels of indirection, you are ready to master the use of pointers in parameter lists. Because C passes all parameters by value, it is not possible to modify any of the input parameters to return. If a function is to return more than one value, it must either return a structure or require that some parameters be passed by address. For example, the following is a function that calculates both the quotient and remainder of a division operation:

```
int mydiv(numerator, denominator, quotient)
int numerator, denominator;
int *quotient;
{
   *quotient = numerator % denominator;
   return(numerator / denominator);
}
```

You call this function with:

```
val = mydiv( top, bottom, &quot);
```

so that, when the mydiv function assigns to *quotient, it will actually store the value in the location pointed to by &quot, which of course is quot itself. While it is easy to see with simple data types, it can be confusing when you start dealing with pointers.

Let's take a simple function that extracts nonblank characters out of a string and keeps track of where it has been:

```
int nextc(stringp)
char **stringp;
{
   while(**stringp != '\0' && **stringp != ' ')
      (*stringp)++;
   return(**stringp);
}
```

This might be good C style, but you can understand it much better if you write it as:

```
int nextc(stringp)
char **stringp;
{
   char *p;
   p = *stringp;  /* get the current position */
   while( *p != '\0' && *p != ' ')
      p++;
   *stringp = p;  /* remember where we were for next time */
   return(*p);
}
```

By removing that one level of indirection throughout the entire routine, you can see that all you are doing is normal string manipulation. The tricky part is getting the parameter in and then saving the result. If you ever encounter a problem in dealing with pointer parameter values, this simplification of the routine should be your first step to correcting it.

Where it can be tricky is in calling the routine:

```
char p[20];
char *cp;
int c;
c = nextc(&p); /* this DOES NOT WORK */
c = nextc(&"this is a string"); /* This doesn't work either */
cp = p;
c = nextc(&cp);   /* This does what we expected */
cp = "this is a string";
c = nextc(&cp);   /* This works also   */
```

What is happening here is that &p means to take the address of the array...but does it? One peculiar aspect of C is that you cannot take the address of an array. In fact, if you pass the array to any function, such as:

```
foo(p);
```

C will automatically pass the address of the start of the array for you. It is because of this that C provides us with a strange syntax anomaly that programmers continue to curse:

```
void confusion(p1, p2, p3)
char p1[10];
char *p2;
char p3[];
{
   ...
}
```

All of the above parameters are passed the same way and act exactly alike. C will automatically take the address of an ►

array for you when you tell it that you are receiving an array parameter, and C automatically does the same for you when you attempt to declare a parameter that is an array. While this may seem like a nice syntax to use, it can be quite confusing reading the code later. For this reason, you are best off using only the form shown as p2.

PUTTING IT ALL TOGETHER

Now that you have seen the traps that C can lay out when using pointers, let's review several important points:

- Remember that a pointer by itself is useless; it is what it points to that matters.
- When in doubt, simplify incoming pointer parameters by dereferencing them at the start.
- Remember that arrays are passed as the address of the first element.
- Avoid using array declarations in parameter lists.

If you think that you are now ready to tackle the world of C pointers, try your hand at the simple test below. Without compiling the code, can you figure out what it does? I promise that it breaks the rules I listed above! Hint: try recoding pieces that seem difficult. (The answer's on page 64.) ■

```
int tryit(val, stringp)
int *val;
char **stringp;
{
    (*stringp)++;
    if (!**stringp) return(0);
    (**stringp)--;
    *val = !*val;
    if (*val) return(**stringp);
    return(' ');
}

void main()
{
    char b[] = "TZHPYVC!OHKLORYCIDSKCKX";
    int j, c;
    char *p;
    p = b;
    strcpy(p+10, "RPNUS!EJFUL\"G");
    j = !p[0];
    while(c = tryit(&j, &p))
        putchar(c);
    putchar('\n');
}
```

John Toebes is coordinator for the Software Distillery and is currently employed at Bell Northern Research. He was a major developer of the SAS/C system for AmigaDOS and has written Amiga projects including BLINK, Hack, PopCLI, Menwatch, and NET:. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on CompuServe (72230,303).*

Font Formats

Bitmap, Intellifont, Type 1, DMF, or IFF DR2D OFNT—which formats should your programs support and to what extent?

By Dan Weiss

THE WIDE RANGE of fonts available for use on the Amiga stands as a point of distinction and confusion. Standard bitmap fonts, ColorFonts, anti-aliased fonts, and a growing array of outline fonts—which should you support in your program and to what degree? Bitmap fonts are the old standbies, but outline fonts are gaining ground in the Amiga marketplace. To make the right decision, you must understand what each can offer you and your program's users.

FONT SPEAK

Before we dive into font format specifics, let's establish a vocabulary of terms. A *glyph* is an individual symbol in a font. Many people would call this a character, which is not entirely accurate. In many languages, a glyph can be more than just a part of a given alphabet. The Chinese written language is made up of pictures called diagraphs that each represent a concept or idea. In the Arabic languages, you cannot say you are correctly printing text unless you can print ligatures, special symbols that represent a group of characters. Even English has ligatures, such as fi and ffi, which are considered crucial to fine typesetting. Because any computer font may be called on to incorporate diagraphs or ligatures as well as characters, it is best just to think of them all as glyphs.

A *font* is a collection of glyphs in a particular typeface and size. In the context of this article, a font is also assumed to be in a computer-usable form for rendering. A *font family* is a collection of fonts similar in basic form, differing only in style. For example Helvetica-Roman is a font, whereas Helvetica-Roman, Helvetica-Oblique, Helvetica-Bold and Helvetica-BoldOblique make up a font family. The terms *bitmapped* and *outline* refer to the computer-usable form in which the fonts are stored.

BITMAP VERSUS STRUCTURED

Bitmap fonts are stored essentially as pictures of what the fonts will look like on the screen. This is very important if the appearance of the glyphs is more significant than their shape, as with ColorFonts. Fonts such as Chiseled Marble, Rainbow, and Metallic are easily created and displayed on the Amiga because of the ColorFont bitmap standard.

The advantage of bitmap fonts in general is twofold. First, the Amiga excels at quickly copying bitmaps, such as glyphs in a font, from one place to another. This makes bitmap fonts fast. Second, because bitmap font glyphs are essentially bitmap pictures, they can be customized and hand-tuned down to the pixel level. Desirable special features, such as anti-aliasing and multicolor details, can be included as parts of the glyphs.

The disadvantage of bitmap fonts is twofold also. The sheer size of bitmap fonts, in terms of disk and memory space required, can get overwhelming. A normal bitmap typeface is often present on the Amiga in at least three sizes, such as 8, 12, and 20 points. Each of these fonts will take up an average of 5 to 10K of disk space, and can go well over 50K including support files. A conservative 10 sets of fonts can consume 300K or more of valuable disk space. The demand for space increases as you add more fonts, font families, or larger fonts.

The reason for keeping so many fonts on hand is to always have one ready that is just the right size and style—which brings us to the biggest limitation of bitmap fonts. The average bitmap font is hand-tuned to look good at a given size. But it looks very poor if it is resized or distorted in any manner. This problem is often referred to as "jaggies," and is common with resizing any bitmapped image.

For comparison, what do outline fonts offer? Like bitmap fonts, outline fonts have certain trade-offs. An outline font only describes the outline of the shape of a glyph and how it is to be filled. The advantage/disadvantage to this is that each glyph is drawn new every time. On the one hand, this allows the computer to draw the glyph to the best of its ability and resolution, thus reducing jaggies. The fonts also take up much less disk space, because one file is needed for each typeface only, not each size.

On the other hand, because the glyphs are only shapes to be filled, the computer must render them, then place them where they belong. The extra step of rendering can produce a very noticeable slowdown in screen display or printing. Font caching offers a partial solution to this problem, however. With font caching, you in effect render the glyphs you are using once, then use them over again as you would use a bitmap font. In this way, you only have the overhead of rendering the glyphs once. But, as with bitmap fonts, you now need a place to keep the bitmap glyphs you created.

Another limitation of outline fonts comes in rendering them at small point sizes. In the case of bitmap fonts, the creator can fine-tune each glyph so it "looks" its best at a given size. The human mind is often referred to as the finest image processor available. Because of this powerful "tool," most people can create readable glyphs with even a very few pixels. Font rendering software is not so lucky. It is limited because it has no idea, beyond the outline data, what the glyph should look like. At larger sizes this is not a problem, but at smaller sizes it is. To overcome this limitation, many manufacturers implement what are known as hints. Hints can take on many guises, but generally they take the form of extra commands that describe certain parts of a glyph. The hints ►

are usually used when either few pixels are available, so as to improve readability, or when many pixels are available, so as to bring out subtle detail.

At the time of this writing, Commodore does not officially sanction any one outline font format. Consequently, several formats are in use. Two major formats that are getting a lot of attention are the Agfa Compugraphic Intellifont and Adobe Type 1 font formats. The Soft-Logik DMF and IFF DR2D OFNT formats are also gaining attention. Let's look at these more closely.

COMPUGRAPHIC INTELLIFONT

The Intellifont format, from Agfa Compugraphic, is used primarily on MS-DOS machines and by the HP LaserJet Series III family of printers. Many facets of the format reflect this heritage, including the font data being stored in Intel processor format. Currently, around 250 hinted fonts exist in this format and are readily available in MS-DOS and, to a much lesser degree, Amiga format.

The main source of information on Intellifonts is a book called *Intellifont Scalable Typeface Format*, which is freely available from Compugraphic. However, as was pointed out by Compugraphic's Tim Christo, the book carries the following warning: "Any attempt to develop an Intellifont reader, rasterizer, converter, or interpreter from information contained in this document is illegal unless expressly authorized by

Hints at Work

"HINTING" IS THE magic that makes high-quality outline fonts look good at any size. The reason for hinting is simple. When you have only a few pixels to work with, knowing which ones to turn on and off is very important. Most times this involves growing or shrinking part of a glyph to make it better fit the grid of pixels it is being drawn on.

Figure 1 shows the difference between the actual outline of an H and the outline fitted to a grid. As you can see, the legs of the H got much wider, and the detail of the serifs (the flared parts at the top and bottom of the letter) was lost. With the H it is easy to see what pixels need to be filled in. More often the letter does not fall evenly on the grid. This can result in having one leg thicker than another or creating a lopsided curve.

To help draw the best possible glyph, the creator of the font will build in hints that control key features of each glyph. In the case of the H, the legs are very important. In the Intellifont system, you might set up a relationship between the width of the two legs to control this (distance A must equal distance B). In this way, when one grows, so does the other. (See Figure 2.) With Adobe Type 1 fonts, you might rely on the standard vertical-stem-thickness hint. By setting this, all of the vertical stems (like the legs of the H) in the glyph will be made the same. (See Figure 3.) □

—DW

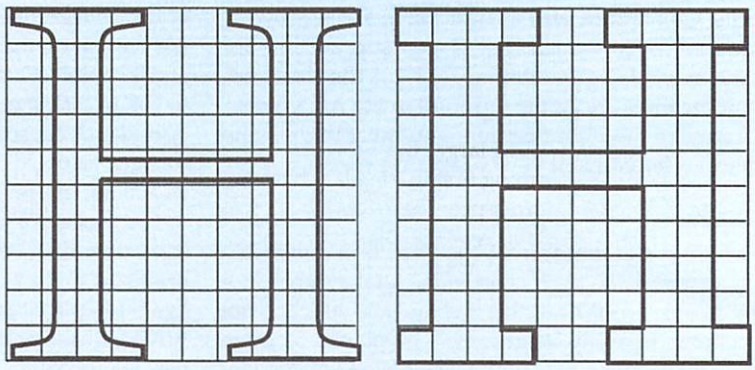


Figure 1: The actual H and an H fitted to a grid.

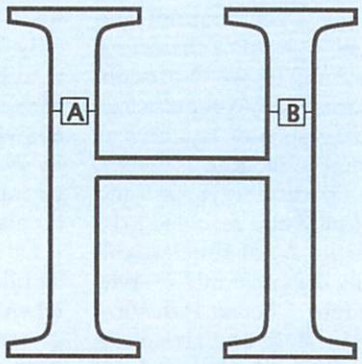


Figure 2: Distance A must equal distance B.

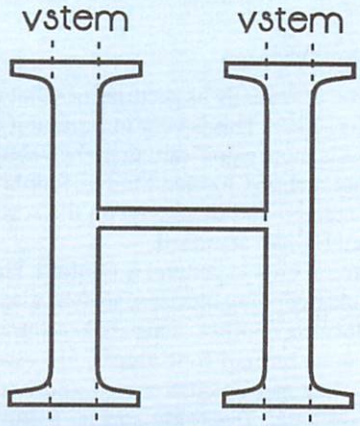


Figure 3: Vertical-stem thickness should be equal.

Agfa Corporation." This means that the data in the book and this article is for informational purposes only.

The first and most notable aspect of the Intellifont format is the way fonts are stored. Intellifonts are straightforward data files, which are very simple for a computer program to read.

Each font actually relies on two files: a Font Attribute file and a Font Display file. Both are structured internally in a similar manner. The Font Attribute file contains the information normally associated with a font metrics file. It also includes general information about the font and instructions for creating multipart and ligature glyphs. The Font Display file contains the instructions for rendering the glyphs. The files are stored using a method called Keyname encoding. This does not mean that the file is encrypted, as is the case with Adobe fonts. Rather, each section is indicated by a code number. The basic format works as follows: At the beginning of each file there are blocks of information, designated by a UWORD, that contain either the actual information, if it is small, or an offset into the file where that data is located. A program can process a block of information or skip over it easily when it is stored in this manner.

The data itself is oriented towards MS-DOS machines and is stored in Intel format. Anything larger than a byte is stored with the lowest-order byte coming first in the file. This is opposite to the way that a Motorola-processor-based machine, like the Amiga, expects the data. As the programmer, therefore, you must "fix" the results of a standard, high-level file read to process the file. One saving grace is that a lot of the data can be tossed away, as it is only there to support HP printers and of little use to anyone else.

When you finally get through the Font Display file to the glyph data, you will find it, to say the least, a bit confusing. The data does not describe curves as arcs or bezier curves, but rather as a collection of points on the curve as well as some other related points. While this does not make the fonts impossible to render, it does add an extra level of complexity.

In addition to the glyph outline data, the Font Display file holds Intellifont hints called Instructions. These are a series of points on the glyph that define relations between different parts of the glyph. For example, to keep an H looking normal, you would want the two legs to remain the same width relative to each other. By setting up a relationship between the two legs, the rasterizer would make sure that they grow and shrink at the same rate. Intellifont hints are based primarily on the concept of relative heights and widths within a glyph. (See "Hints at Work.")

ADOBE TYPE 1

Adobe Type 1-format fonts have their origin in PostScript fonts from Adobe. At first, only Adobe knew the special secret format of these fonts. They had a distinct advantage over other PostScript fonts in that they were much smaller and had hinting built into them. Adobe jealously guarded the secrets of this format, encrypting the contents of each font to prevent examination. Under intense pressure from the publishing community, on many computer platforms, Adobe finally released the details of the format. It is now widely supported on many platforms, with thousands of hinted fonts available on the Macintosh and MS-DOS platforms, and many coming to the Amiga.

Like the Intellifonts, there is a single source of information on Type 1 fonts—*Adobe Type 1 Font Format*, available from Adobe Systems Incorporated. Unlike Compugraphic, Adobe maintains that their format is open to the public. The notice in the Type 1 book merely warns that the information contained within is subject to change. This openness is a mixed blessing to Amiga developers, but for now let's examine the structure of Type 1 fonts.

Adobe-format fonts are radically different from Intellifont fonts. Where Intellifont fonts are primarily data files, Type 1s are PostScript programs. This distinction is very important to creators of such files, because in the U.S., fonts are not copyrightable, but programs are. (Note: Compugraphic also claims this distinction, but I would disagree. Time for the lawyers.) Because of the program nature of a Type 1 font, the code that will read the file must think like a PostScript interpreter. This is not an easy task.

An added problem for a font renderer is that, for space and security reasons, all Type 1 fonts are encrypted and first must be decrypted before they can be used. The encryption process is actually in two parts. First, the font instructions, which are normally in standard ASCII text, are converted to numerical codes, and all numbers are converted into a special unsigned multibyte format. After the commands have been reduced to a purely numeric format, then they are encrypted. A side result of this is that the issue of processor byte ordering (Motorola versus Intel) disappears, because the font information must be decoded on the byte level.

On the up side, once a font has been decrypted (which can actually be done on the fly), the font is pretty straightforward to render. Font outlines are described using standard straight lines and bezier curves. Hints are included in the font program, but do not need to be executed to properly render the font.

The AFM (Adobe Font Metric) file, ironically, is coded in perhaps the worst way for a computer; it is a plain ASCII text file. While this makes it very easy for anyone with a text editor to change the file, many computer languages do not have native facilities for converting text to numerics. Depending on your need for kerning and width data, you may not even want to mess with the AFM file, as it only contains metric data.

The hints included in Type 1 fonts take a different approach than those in Intellifont fonts. They make some generalizations and offer some solutions to very specific problems. Like Intellifont, the Type 1 standard can establish standard widths and heights, but, in general, only one of each per glyph. This tends to make all parts of the glyph the same, removing some of its typographic distinction. Type 1 hints go beyond Intellifont hints in other areas though, such as defining certain glyphs to be bold at small sizes and defining subtle curves often found in serifs. (See "Hints at Work.")

DMF

Freely available from Soft-Logik, the DMF format is stored in a data file like the Intellifont format, but is much less complex in its implementation. These fonts are not encrypted in any manner and use standard lines, arcs, and bezier curves to describe glyph outlines. Although this format lacks hinting, it is very easy to use. I would not recommend DMF as the core font choice for a commercial program. However, it makes an excellent choice for a public-domain program or a commercial program supporting more than one font format.

IFF DR2D OFNT

The OFNT format, from Stylus (Taliesin), is part of the new IFF DR2D structured drawing format, and has an IFF struc- ►

ture itself. I hope this format gains wider acceptance in the future. At this time, however, few programs know how to render fonts in this format, and few such fonts exist. The format itself is documented in the IFF DR2D specification. It is nonencrypted and supports bezier curves. Although tightly associated with IFF DR2D, you can use this font in any situation. Conversely though, you are not required to use only this format with IFF DR2D files.

TRUETYPE

TrueType is a format that was developed at Apple as part of their System 7 operating system enhancements. The format is very ambitious in its scope. Its hinting system outstrips any current system. The hinting was specifically designed to be a superset of existing hinting systems. This format will also become the native format of Microsoft Windows. The problem with the format is that it is a bit too ambitious. TrueType is very complex, and does not even run that well on its native platform, the Apple Macintosh. At this time I would not count out the TrueType format, but I do not think it has any serious bearing on the Amiga.

WHICH SHOULD I CHOOSE?

In the process of writing this article, I interviewed representatives from Commodore, Compugraphic, Adobe, Soft-Logik, and Stylus. The "inside" word they gave me will probably (unfortunately) have more influence on your choice than the true merits of each format.

Andy Finkel of Commodore said, "Transparent support for Compugraphic fonts is in version 2.0.4 of the OS." Andy also stated that future versions of the OS will feature more Compugraphic support. This would indicate a strong interest on Commodore's part in Intellifont. This also means that many programmers can merely wait for the promised support and do nothing for now.

The problem, of course, with waiting for any feature upgrade is that you are never sure when it will appear. According to Andy, the first phase of support is in the pipeline and will be available very soon. This phase is transparent, meaning that any program can use the standard font calls and get a bitmap font at any size from the outlines. For many people, this is all that is needed. The second phase is scheduled for the end of the year and will allow programmers to make calls to the Compugraphic code and get back outlines.

Some people may need to get at actual outlines sooner than that. Gold Disk and Soft-Logik have already incorporated Intellifont support into their programs. If you are looking to go that route, you should contact Dale Hubbard or Tim Christo at Compugraphic. According to Christo, Compugraphic will supply Intellifont rendering code to any Hewlett Packard (HP)-registered software developer free of charge. "We are in the business of selling fonts, not rasterizers," said Christo. The code is in generic C, as he puts it, and well documented.

But only about 250 hinted Intellifont fonts are available, whereas several thousand hinted Adobe Type 1 fonts are available. Unfortunately, Adobe has made it clear that they have little interest in the Amiga. According to David Downing of Adobe Systems Inc., "We have no plans in the works for the Amiga." He also indicated that Adobe has its hands full with platforms it currently supports. This means if you want to use Type 1 fonts, you will have to roll your own.

As for DMF and IFF DR2D OFNT fonts, the story is similar. The creators of these fonts will be very helpful, but in the end you will have to write your own rasterizer.

Even on other platforms, like MS-DOS and Macintosh, the question of which format is best is not clearly answered. My suggestion is to go with the font that will best meet your needs at this time. For most people, the promised OS support for Compugraphic fonts is enough. If you just want outlines and don't want too much work, then DMFs or IFF DR2D OFNTs are good choices. If you need a large body of fonts or want to keep in step with the rest of the computer marketplace, then Adobe Type 1 is the only choice. ■

Dan Weiss is Vice-President of Research and Development at Soft-Logik Publishing. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or on BIX (dkazmaier).*

Bibliography

All the font formats mentioned above are documented in publications available to the public. Some are free, and some have a nominal charge. Below is a list of the publications and how to get them.

Adobe Type 1 Font Format
Adobe Systems Inc.
1585 Charleston Road
Mountain View, CA 94039
800/833-6687
$14.95

Intellifont Scalable Typeface Format
Agfa Compugraphic
90 Industrial Way
Wilmington, MA 01887
OEM Technical Support Group
508/658-5600 x2218
Free

Soft-Logik DMF Font Format
Soft-Logik Publishing Corporation
11131F South Towne Square
St. Louis, MO 63129
Dan Weiss, VP Research & Development
800/829-8608
Free

IFF DR2D OFNT File Format
Stylus Inc.
PO Box 1671
Fort Collins, CO 80522
Jeff Blume, VP Operations
303/484-7321
Free

TrueType Spec - The TrueType Font Format Specification
Apple Computer Inc.
800/282-2732
Product # M0825LL/A
$30

Manual Training

Don't cripple a great product with weak documentation. Follow these tips and produce an equally great manual.

By Daryell Sipper

OTHER THAN GOOD design and programming, the only tool you have for creating a user-friendly product that sells is your program's manual. Many customers choose between similar software products based on documentation. A thorough manual will also reduce your customer support efforts, saving you time and money. It can contribute to positive magazine reviews, further advertising your product. Most important, a good manual helps create loyal customers.

The keys to producing a quality user manual are:

- Good planning and preparation.
- Good writing style.
- Organization.
- Providing an abundance of information and detail.
- Using an easy-to-read format.
- Following the basics of good manual design.
- Including the manual in your production schedule.
- Having the proper attitude about producing the manual.

To ensure that your manual is truly user-friendly, you must consider the least knowledgeable user as well as the most advanced. Use plain English instead of technical talk, acronyms, and jargon. You should impress the user not by how much you know, but by how easy it is to use your manual (and therefore your product). Don't get wordy—use short, easy-to-read sentences. Write from the user's viewpoint, not the programmer's. Include many examples. Don't be afraid to use the active voice and present tense; talk to the reader.

AUTHOR, AUTHOR?

One of the first steps in your planning stage is deciding who should write the manual. Should you, the developer, write the manual or should an independent writer? Both answers have merit. You might want to write the manual to maintain complete control of the project and to minimize costs. You also understand the software and know how it functions. Unfortunately, this is a reason for you not to write—you are too close to the project. Although you may be an expert in creating software, you may not be an expert in communications.

While an independent writer may have better communications skills, he probably knows little about your software. He has to be broken in, which takes time. Then again, that's good, because the writer brings in a fresh, objective attitude. The writer sees the product as a user, not as a developer. The writer also indirectly serves as a beta tester because of the independent writing process.

Another solution may be a cooperative effort, with each doing what he's best at. You assemble the initial outline and notes. The writer picks up from there, does the hard-core writing and organizing, then polishes everything.

If you decide to write the manual, you should begin by compiling your reference library and support software. I recommend a dictionary (*Webster's New Universal Unabridged Dictionary*), a thesaurus (*Roget's International Thesaurus* or The Electric Thesaurus from Softwood Corp.), style guides (*The Elements of Style*, Strunk & White; *The Elements of Grammar*, Margaret Shertzer; *The Goof-Proofer*, Stephen Manhard; *Technical Writing*, Bly & Blake; *The Technical Writer's Handbook*, Matt Young), and a word processor with spell-checking, outline, table of contents, and index-generation features (WordPerfect from WordPerfect Corp.). An electronic grammar and style checker (Proper Grammar from Softwood Corp.) and a desktop-publishing program (Professional Page from Gold Disk) are also useful. Electronic grammar and style checkers are excellent proofreaders for a variety of common mistakes (typos, double words, homonyms, punctuation, mixing tenses). Do not, however, rely on them totally. The program has no idea of your intent, and you as the writer must be the final judge. If you have not written for a while or are uncomfortable with writing, consider taking refresher classes.

THE TIME IS RIGHT

You are still not ready to begin writing the manual. You should next develop a schedule. A basic schedule will look similar to the following:

1. Analyzing your audience and defining the manual's objectives.
2. Drafting an initial outline.
3. Researching and collecting information.
4. Developing a detailed outline.
5. Writing the first draft.
6. Illustrating the manual.
7. Editing and rewriting.
8. Testing and proofreading the draft.
9. Editing and rewriting the draft.
10. Designing the manual.
11. Testing and proofreading the manual draft.
12. Editing, rewriting and redesigning the manual.
13. Writing the table of contents and index.
14. Assembling the final manual.
15. Final testing and proofreading.
16. Final editing.
17. Publishing.

When analyzing your audience for step one, ask yourself:

- Who is going to use this product?
- What will be their knowledge level?
- What is their experience? ►

- What do you expect of the user?
- Who will be the least knowledgeable user?
- Who will be the most advanced user?
- What reading level is appropriate?
- What writing style is appropriate?

WHAT'S INSIDE?

Before writing your outline, determine your manual's organization. Most manuals should contain:

1. Title page.
2. Copyright page: Include information for customer support, the software version/release number, program disclaimers, warranties, and credits for programmers, authors, beta testers, spouses, children, pets, and so on.
3. Preface: Explain the product—what it is, who it's for, what the basic features are, and the benefits and potentials for the user.
4. Table of contents.
5. Introduction: Explain how to use the manual, including a basic overview of the program and the basic terms used.
6. Getting started: State your assumptions about the user's background or expertise, what hardware and software are required, the user installation procedures, and how to begin using the product.
7. Tutorial or main body of the manual.
8. Reference section.
9. Appendixes.
10. Glossary.
11. Index.
12. Reference cards and, possibly, keyboard templates.
13. User feedback card: Don't incorporate this with the user registration card. The user needs to register immediately, but usually won't be able to offer constructive feedback for some time.

"Keep your writing simple and keep everything in bite-size chunks."

Your writing efforts should follow a basic four-step process. First you prewrite—that is, you assemble your ideas in an outline. Then you expand that outline by writing the text. Editing (and rewriting) is the third step. Formatting (design/layout) is the last step.

Your detailed outline should be well organized, logically structured, and provide for a natural flow of information. Once your outline is completed, you can then start your first draft. Writing isn't easy. Don't get frustrated. Follow your outline and don't worry too much about the editing or the format. For now, getting the ideas down is most important.

When you finish the initial draft, you should revise and organize everything. Keep your writing simple and keep everything in "bite-size" chunks. Remember to keep one thought per sentence, one sequence of thoughts per paragraph. Talk to the reader, using an active voice and the present tense.

Completing the first draft is a monumental accomplishment, but you are far from finished. Now you need to consider illustrating the manual. Illustrations serve several purposes. They help break the monotony of reading, they help support your text, and they pictorially explain your text.

There are several types of illustrations. Line-art and halftones are the most popular. Line-art consists of only two solid colors with no shades or tones. An electronic schematic drawing is an example of line-art. Halftones have shades or tones. A scanned photograph is an example of a halftone. The best illustration you can provide of your program is an actual screen shot. A screen capture will be much more convincing than a hand- or computer-drawn simulation of your program. Once saved in a standard file format, it can be imported into your page layout program.

Once you have chosen your illustrations, you should edit and rewrite your draft. Remember those words. You'll spend much of your time editing and rewriting before completing your manual. Follow the basic rules of good grammar and style. Give particular attention to punctuation and common grammatical errors. *The Elements of Style* and *The Elements of Grammar* are excellent handbooks to keep at your side.

Now you can send the draft to your beta testers. Develop a testing checklist for them. They should check for such things as style consistency, grammar, spelling, comprehension, missing information, wordiness, pagination, and misinformation.

You've entrusted these people to help you develop your product. Be prepared for their comments and try to have a business-like attitude. Writing is very personal and it's easy to let your tester's comments upset you. They will find mistakes. The mistakes will vary from simple typos to conceptual misunderstandings. Don't take their comments as an attack. Just remember, the more mistakes they find, the fewer your customers will find. That's good! When your testing crew finishes their proofreading, edit and rewrite again.

MANUAL DESIGN

Now that you have tested, corrected, and rewritten your draft, you can start considering your layout and design. With the very capable desktop publishing programs currently available you could design the manual yourself. Be forewarned: These programs have long learning curves. They are fairly easy to use, however, once you have the hang of them. If you do not have the time or experience, seriously think about hiring outside help. Page layout and document design is a specialized field that deserves entire books and study of its own.

If you decide to plunge in, there are some basic rules to remember. The design should be pleasing to the reader's eyes. No matter how technically correct your manual's contents might be, its appearance will determine the user's initial reaction to using it.

Use white space effectively—do not attempt to fill up a page just to save a few pennies in paper. A good rule-of-thumb is to reserve, as a minimum, roughly 10% of the page length for both the top and bottom margins and roughly 12% of the page width for the side margins. Leave enough space for the user to scribble notes. If you choose to use columns, they should be roughly one to two alphabets (a–z) wide, or about nine to ten words.

Keep your text in small "bite-size" chunks. That means keeping sentences short. Keep your paragraphs short. Insert blank lines between your paragraphs. Separate the sections

by larger than normal spaces. Boldface the section titles. Avoid page breaks during instructional sequences.

Use both upper- and lowercase letters. Use blocked or ragged-right justification, depending upon your margins. Do not use a "typewriter" font. Use a typeface that's pleasing to look at. Do not strain your reader's eyes; use a type size that's large enough to read without a magnifying glass.

Use a serif font for the basic text. Normally, body text is 14 points or smaller. Headlines and display type will be larger than 14 points and usually boldfaced.

Ensure that your illustrations and graphics don't confuse your reader. Number them and label them with captions. If the illustration is larger than half a page, consider putting it on a page by itself. Keep your illustrations close to the associated text, not three or four pages away. Use actual screen shots of your program.

Use numbered or bulleted lists instead of cramming a series of items into a paragraph. Use chapter or section markers. Ensure that the page size is easy to handle; 5½" x 8½" and 7" x 8½" are popular sizes. Start new chapters on right-hand pages. Above all, ensure that the design is consistent throughout the manual.

You should distribute copies of your design draft to your testing family for comments. Again, be prepared. They will find the mistakes you missed. But that's what you hired them for, isn't it? Based on their input, you should once more edit, rewrite, and modify the design of the manual.

"You alpha, beta, and gamma tested your software. Your manual should be tested, too."

REFERENCE MATERIAL

By now your basic manual is complete. It's written, illustrated, paginated, and designed. What's left? Two of the most important sections—the table of contents and the index. I cannot emphasize enough how important these two sections are. The more information these two sections contain, the happier your customers will be. It's that simple.

Developing a table of contents is a straightforward process, even without the assistance of your software. Unfortunately, developing an index is not. The current state of Amiga page layout programs does not help, either. None directly support the generation of indexes (or tables of contents or lists). You'll have to use your word processor to develop your index. Copy your manual's text back to your word processor. Adjust the page breaks so they match those of your page layout program, and then create, edit, and verify your index.

If you have a large or complicated manual, develop an index with subtopics as well as the original topic. For example, don't just list all the places where the word "page" is used. Break that down into "page, breaks" and "page, length," and so on. If you really want to impress your customers, cross reference everything.

You might prefer to develop the index as you write the manual. Waiting until the entire manual is written before creating the index makes it an overwhelming chore, especially if the manual is exceptionally large. Developing the index on a section or chapter basis can be less intimidating. The drawback to this method, of course, is that subsequent rewrites will change your index. Take the extra time to develop an exhaustive and comprehensive index. Your customers will be thankful and, ultimately, so will you.

Your customers may also appreciate a detailed reference section, an appendix section, or on-line documentation. In them you can explain certain aspects of your program (such as a listing of possible error messages, some technical theories, or notes about your program) in more detail than in the main body of your manual. On-line documentation (manuals, addendums, read-me files) and help screens are tremendous tools that can help create a user-friendly product. Much of the material can be taken directly from your manual. This might mean some additional programming and testing, but it's part of your overall documentation (and testing) process. Don't forget a glossary, reference cards, and keyboard templates.

FINAL TOUCHES

Now you're ready to assemble your manual. Again, make only a few copies for your dedicated testing crew. They now have the task of performing the final proofreading. No one will look forward to this, but it must be done! Remember, you still have to edit and correct everything based upon their response, including the table of contents, references to page numbers, and the index. Extend your testing checklist to include these items. No one said this was easy!

Before you send your final copy to the printer, you should consider usability testing. Find people who have never seen your product or manual. Ask them to test your manual. Why not? You alpha, beta, and gamma tested your software. Your manual should be tested, too.

How about the final printing? Your dot-matrix printer is fine for proofing, but not for final copy. Take your manual, in PostScript format, to a professional typesetter. You should do so even if you own a 300-dot-per-inch laser printer. Consider using your laser printer only if there are no halftone (photograph) illustrations and you are selling just a few copies of your product, such as in a vertical market.

Loose-leaf and spiral bindings are your best choices for your manual's physical assembly. The least favorite are hard binding and stapling. These two are the least expensive but the most frustrating for users. Users need their hands available to operate their keyboard and mouse, not to keep a manual open. If you envision many upgrades and enhancements, seriously consider loose-leaf so users can easily add extra pages. Otherwise, stick with spiral binding. Use stapling only for the smallest of manuals.

Above all, remember that your manual is part of the total product package, not a last-minute by-product. Establish a system that works for you. If you prefer to develop your index as the manual is being written, do so. If you prefer to choose all of your illustrations before writing, do that. The keys to the whole process are commitment and organization. ■

Daryell Sipper has owned and used an Amiga for over five years. He is owner of Midwest Desktop, a writing and documentation service. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

Arcade Elements

Pull together the building blocks for a top-flight shoot-'em-up.

By Tony Scott

DESPITE THE APPARENT simplicity of the results, coding an arcade game can be a daunting endeavor. Even the most simple game incorporates most of the Amiga's special abilities. Graphics, sound, and interfacing to real-world hardware must all come together in a small, fast, and interesting package. Scattered throughout dozens of books, manuals, and tutorials are the basic components necessary to construct a complete game. Finding these tiny shreds and integrating them into a complete program can be frustrating at best. While by no means comprehensive, this article illustrates how to combine many of these elements into a coherent whole.

Be warned: Many of the methods used here are *multitasking-hostile*, basically taking over the machine (or at least preventing Intuition from seeing any input events). I do not recommend many of these methods for anything except games, because such actions are generally considered unacceptable behavior for applications software. If you are considering writing a multitasking-friendly program, see the Improvements section below.

THE TYPES

Before we plunge into the SAS/C example program (game.c in the disk's Scott drawer), let's establish the structure variables we need. For the joystick data we have:

JOYDAT

x	**Left/right status of the joystick**
y	**Up / down status of the joystick**
fire	**Status of the fire button**

For game variables we use:

OBJECT

x,y	**Cartesian coordinates of the object**
dx,dy	**Speed of the object**
rot	**Current rotation of the object**
thrusting	**Thrust status (on or off)**
lastshot	**Time of object's last shot**

From now on, following along with the game.c source may be helpful. (Manx C users take note: Because the compiler does not support the _chip keyword, you will need to allocate a block of chip RAM, copy the graphics data into it, and use the allocated data rather than the data defined in the program.)

VIDEO: THE HARD WAY

The most important part of any game is the visual component. The Amiga's hardware provides two methods to achieve smooth, fast animation. The first and most popular method, using hardware sprites and bobs, has been covered extensively. The second is more complicated and often spoken of in hushed tones of awe and mysticism. This conjury, called *double buffering*, is no more than simple slight of hand. Using hardware sprites would have been much easier for this particular example, but to illustrate several key concepts I chose Blitter calls and double buffering. Double buffering is a way of switching between two different display areas so that the graphics rendering is always hidden from the viewer. This prevents flashing, flickering, or transparent animation.

The procedure is relatively simple. First, create a basic display consisting of a view, its RasInfo structure, a viewport, and a bitmap, along with a spare bitmap for later use. After initializing and linking these structures, calls to MakeVPort(), MrgCop(), and LoadView() bring up the display and compile the appropriate Copper lists. For the double-buffering part of the display, save a copy of those Copper lists, then force the system to make a new set for the other bitmap. This new set is created by setting them both to NULL and the RasInfo-> BitMap pointing to the spare bitmap, then calling MrgCop() again. Create a rastport to let you use Text() and Blitter masking functions with the display. This rastport is constantly changed so it uses the bitmap that is not currently being displayed (i.e., savebm). Figure 1 illustrates the double-buffering process.

THE SCORE

To avoid converting numbers to strings, we start the score at 48 (the ASCII value of 0) and go up to 58 (the ASCII value of 9), allowing us to use the Text() function directly on our score data. This method limits us to scores ranging from 1 to 10, but saves lots of code.

When the program calls ShowScore(), the routine displays both players' scores, each in the proper color. Because we have two displays (both of double buffering's bitmaps) to worry about, we call SwapDisplay() and display the score a second time. This insures that both displays show the same score and there is no flickering.

THE GAME TIMER

To keep track of game time, we create our own minitimer. This minitimer is nothing more than an unsigned WORD incremented for each frame. Because we are rendering a new frame about 60 times per second (the vertical blanking speed), the counter will not return to 0 for about 828.5 days. To determine this, divide 4,294,967,295 (the max value for a ULONG) by 60 for the maximum number of seconds, then

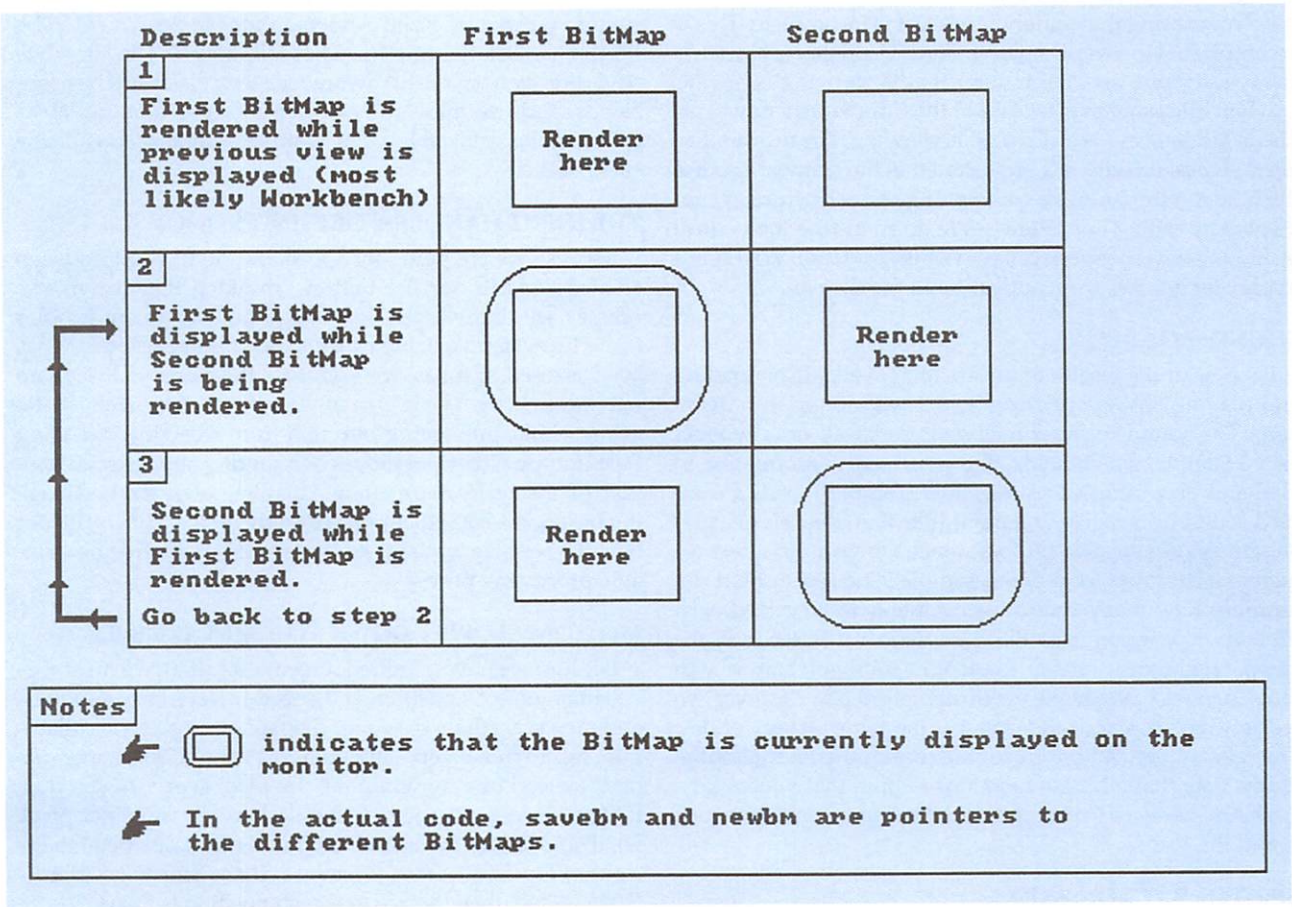


Figure 1: A graphic representation of double buffering.

convert seconds to days. 828.5 days should be plenty of time for a single game. Using this timer we can regulate such elements as how often a missile can be shot and how long a game has lasted.

PRETTY PICTURES

DoImagery() sets up a 16×210-pixel, two-plane-deep bitmap containing the imagery for each of the various graphics objects that will be rendered during the game. This bitmap is divided into 21 16×10 frames, each of which is described below.

Frames	Description
0–7	The ship's image in each of its eight facings.
8–11	The sun image in each of four positions.
12	The missile image (only one).
13–20	The ship thrust in all eight facings.

Because both planes of the bitmap are identical, you can achieve three different colors for each image with only one bitmap by masking certain planes when blitting this imagery to the display. These are:

Plane 0	Plane 1	Color	Mask
OFF	ON	Blue	1
ON	OFF	Red	2
ON	ON	Yellow	3

By using BltMaskBitmapRastPort() with the first bitplane as a mask we can blit only where the ship is located, instead of the full 16×10-pixel square. If we want multicolor objects, we must create an additional bitplane to serve as a mask and another eight frames for the second ship image.

BEHIND THE SCENES

Actual rendering is always done to the savebm bitmap, while the newbm bitmap is begin displayed. An overview of the rendering process looks like this:

1. Clear the bitmap that is not showing.
2. Blit ships, sun, and missiles to the hidden display.
3. Hide the current display, and show the hidden display.
4. Make sure the rastport uses the hidden bitmap.
5. Load the new display and Copper lists.
6. Go back to step 1.

We'll discuss the process in more detail as we examine each routine individually below. (For a flowchart depiction, see Figure 2.)

CLEARING THE BITMAP: SAVE THOSE SCORES!

When clearing the bitmap, we leave the top ten lines alone so that we do not have to redisplay the score 60 times a second. To do this, we use BltClear() on each bitplane of the hid- ►

den bitmap starting 400 bytes from the beginning (10 lines at 40 bytes per line).

THE SWAP

As mentioned above, we use two bitmaps to prevent the user from seeing the rendering while it is incomplete. This is accomplished in SwapDisplay(). The bitmap that is hidden is always savebm, and the bitmap that is showing is always newbm. The same may be said of the Copper lists. SaveCprList is hidden, and NewCprList is showing. For text rendering and masked blits, it is important that the rastport also use the hidden bitmap. After we rearrange our structure, we redisplay by calling LoadView(). We do so during the vertical blanking period between calls of WaitBOVP() and WaitTOF() to prevent glitches from appearing in the display.

SOUND: AGAIN?

Because of the wealth of information available on loading and playing 8SVX sound samples, I will not go into detail here. The sound routines included are based on a typedef called Sample, which holds all the necessary information to load and play sampled sounds. LoadSample() loads a sampled sound from a given filename. PlaySample() plays a given sample on a specific audio channel. FreeSample() frees up memory allocated for a given sample. The repeat field determines how many times this sample is to be cycled each time your program calls PlaySample(). If it is set to 0, the sample will repeat forever (or at least until you stop it with StopChannel()). We make the thrust sound play "forever" so we only have to start it and stop it as the game player decides. This allows the sample to sound continuous, not choppy. Please note that all these functions require that you call the GrabAllChannels() routine first to avoid a visit from our friend the Guru.

SPRITES: BUT YOU SAID...

No, we are not using sprites in this example. We call OFF_SPRITE to get rid of the pesky Intuition pointer. At the end of the program we politely call ON_SPRITE, in case someone wants to use the mouse pointer again.

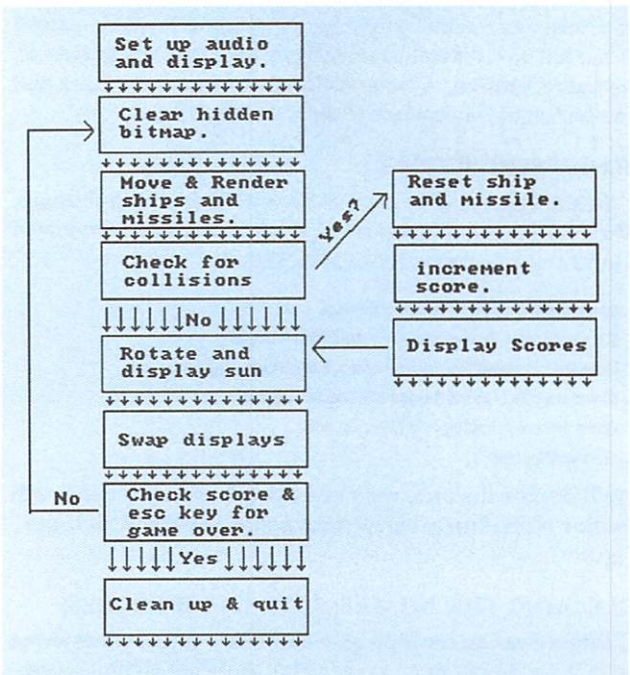


Figure 2: The game's basic flow of control.

THE JOYSTICK: OH JOY!

The joystick routine provided is based on the assembly routine written by Rhett Anderson for "In Search of...The Perfect Joystick Routine" (p. 34, April/May '91). In short, we read directly from the hardware registers, mask out the proper bits, and use these values as an index to look-up tables called tablex and tabley. For complete details, consult the earlier article.

THE INPUT HANDLER: GETTING STINGY

Because we are going for speed and don't want Intuition to meddle with our fire buttons, thinking they are mouse presses, we take over the flow of input.device. In our handler (called KeyHandler), we check to see if the Escape key has been pressed. If it has, we terminate the game and clean up our mess. If the event was anything else, we swallow the event whole, preventing Intuition from receiving it. Calling RestoreInputChain() removes our handler and relieves Intuition of its sensory deprivation. This alternative method to using Intuition's IDCMP port has the advantage that you do not have to perform a Wait() or a busy GetMsg(), freeing valuable processing power.

MOTION: SLOW IT DOWN TO SIMULATE SPEED

Because we have a limited screen resolution, we must represent an object's position and speed in very large numbers and then scale them down for display purposes. This allows our ship to move very slowly (at one or two pixels per second) instead of a minimum of one pixel every $^1/_{60}$ ($^1/_{50}$ on a PAL machine) of a second. Our scale is 100 units per pixel. So, if ship[1].dx equals 20, then it will move one pixel to the right every five frames. Likewise, if the ship's position is (10567,6087), then the actual screen position is (106,609).

GRAVITY: HEAVY MATH

Because we want the sun to have gravity, we must simulate how the ships move in relation to the sun. Actual gravity is calculated by $m \div r^2$, where r is the distance between the two bodies, and m is a constant dependent upon the mass of the two bodies. In reality, this equation works quite well (after all, we don't go spinning off into space), but for our game, the results are far from enjoyable. Using this equation, the ship is pulled too hard within about one inch of the sun and not at all near the edges.

To compensate, we use the equation $168 \div r$. This makes gravity a linear function and closer to what a game player expects. Why 168? I chose 168 so that at the farthest possible edge (320,200) the "pull" would be at least 1 unit ($\sqrt{(320\times320)+(200\times200)} \approx 168$). The pull is how fast the object is drawn towards the sun and is distributed proportionally according to how far apart the x and y distances are. Currently, floating point math is used, but with the addition of an integer square root function, integer or fixed point numbers would be a better choice.

SHIP MOVEMENT: HARD TO PORT

In the DoShip() routine, we move, rotate, thrust, and fire our ships. First gravity is applied, and then the joystick is read.

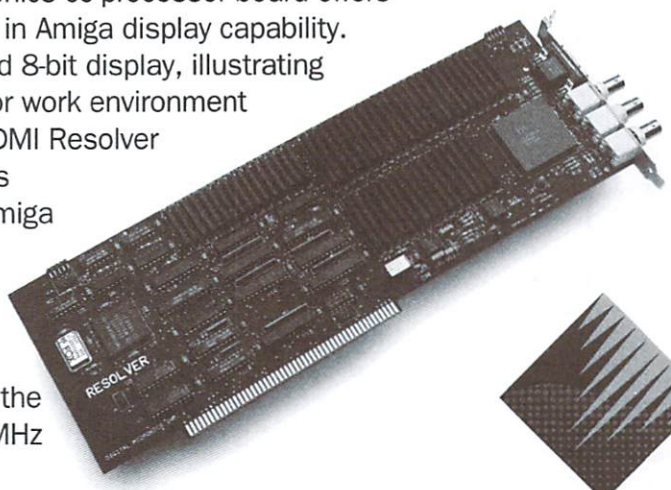
If the joystick is pressed left or right, we increment the rotation of the ship. Each increment is only one quarter of what ►

Don't be Fooled by any other Solution. 1280x1024 Resolution.

DMI Resolver™

- 1280x1024 Resolution
- 8-bit Color Graphics
- 16-million Color Palette
- 60MHz Processor
- Programmable Resolution

The DMI Resolver™ graphics co-processor board offers a new dimension in Amiga display capability. Shown above is an unretouched 8-bit display, illustrating the 1280x1024 resolution color work environment provided by the Resolver. The DMI Resolver boosts the display and graphics processing capabilities of all Amiga A2000 and A3000 series computers, under both AmigaDOS and UNIX operating systems. Not to be confused with a frame buffer or grabber, the Resolver is a lightning fast 60MHz graphics co-processor.



Whatever your application – desktop publishing, presentation graphics, animation, 3D modeling, ray tracing, rendering, CAD – let the Resolver move you into a new realm of resolution and workstation quality display.

Digital Micronics, Inc.
5674 El Camino Real, Suite P
Carlsbad, CA 92008
Tel: (619) 431-8301 • FAX: (619) 931-8516
Call for more information and the dealer nearest you.

Circle 6 on Reader Service card.

is required to actually turn the ship to its next facing. We do this to make ship control less touchy, in much the same way that our movement works.

If the fire button is pressed and a missile has not been fired lately (in the last 15 game ticks), we set the ship[].lastshot equal to the current gamecounter and set its speed relative to the ship's speed. This relative speed means that the player can fly in one direction, shooting backwards, and have his missile follow him. The missile is rendered with DoMissile().

If the joystick is pressed up, then we have a lot to do. First, we find values in the look-up tables thrustx and thrusty and adjust our speed according to which direction the ship is facing. To visually show that the ship is thrusting, we blit the ship image and the image of the thrust. This requires the BltMaskBitMapRastPort() function, because we do not want the thrust imagery to destroy the ship imagery. Next we find out if the ship was thrusting last time. If it was not, we start the thrust sound, which will play until it is turned off. If the joystick was not pressed up and the ship was thrusting last time, we turn off the thrust sound.

MISSILES AWAY

The procedure DoMissile() moves and displays each player's missile. First, it determines whether or not the missile is on-screen by looking at the ship[].lastshot and comparing it to zero. If the value is not equal to zero, then DoMissle() moves the missile by its speed, checks to see that it is still on-screen, and then blits the missile image to the proper hidden bitmap.

When the missile finally goes off the screen, the routine sets the ship[].lastshot to zero, preventing the missile from being rendered until the player shoots again.

THE COLLISION: CRASH!

Our game would not be much fun if the objects did not crash into each other. For simplicity, only the ships and the opposing player's missile are checked for collision. If a collision is detected (the missile lies within the 10×10-pixel area of the ship), the score is increased and shown, the boom sound is played, and the destroyed ship is reset back to its home position with ResetShip().

IMPROVEMENTS: DON'T STOP NOW

For the sake of simplicity, I left many things out of this program. The most obviously absent is error checking. The program does very little error checking, which is very bad form. A possible exception to this is if you are requiring the player to boot your software with nothing else running. It is still not a good idea, but a very common practice with many game designers. Add error checking to your own version.

A few other improvements would be:

- Open an actual Intuition screen and allow input events to pass through to Intuition, letting the program coexist with other programs more easily. You should still stop mouse-button events, however, when the game screen is active. This method would prevent Intuition from giving you those events, yet let Intuition pass them through when other programs needed input.
- A more accurate collision detection scheme could do a logical AND of the ship's and the missile's images to determine if a collision occurred. This would allow many more close calls in which the missile just skimmed past the ship.
- More than eight facings would make for a better game. This would entail a larger databm and look-up tables.
- Collisions with the sun should be fatal, as should collisions between two ships.
- Missiles could bounce off the edge, or both ships and missiles could be allowed to "wrap around" the screen, such that if an object moves off one side of the screen it reappears on the opposite side.
- Hyperspace or shields could be added when the player pushed his joystick down.
- An additional couple of players could be added by using the keyboard handler in addition to the joysticks.
- Multiple missiles (from the same player) could be allowed on-screen simultaneously.
- A computer-controlled ship would be nice. It would have to worry about its facing for both firing and for moving itself around. It should try to avoid missiles, and keep itself from flying into the sun (if it were dangerous).

YOU'RE ON YOUR OWN

Given these methods, you should be well on your way to designing the hottest, most amazing game of the decade! These procedures are the building blocks for many advanced features found in most commercial packages. Add to them, change them, experiment, and enjoy. The real challenge to designing a good game is developing a new and unique concept that grabs and holds the player's attention, but I'm sure you've already got that much, right? ■

Tony Scott is President of KarmaSoft and the programmer of Power Pinball. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

Multitasking in Amiga Basic

From p. 29

end, thus returning control to Workbench. If the close gadget has not been clicked, the program stays within the loop, politely waiting for the 60-second event specified by the ON TIMER(60) command. When this occurs, the ShowTime routine is executed and the current system time is displayed in the window via the TIME$ function, then execution returns to the SLEEP loop to wait another minute for the next event.

To illustrate the function of the SLEEP keyword, omit it from the program listing and run it again. While it runs try using another concurrent application and you should see the difference in the form of sluggish response in the new application. This is caused by the CPU-hogging loop that no longer has SLEEP to provide proper multitasking protocols.

Through the use of the SLEEP keyword and event trapping with the ON event GOSUB commands, you can program many types of multitasking applications with Amiga Basic. Using the same programming principles and either the RUN or CHAIN command, you can also use one task to launch another and both can run at the same time Amiga Basic programs will not run with the interpreter on Amigas equipped with the 68030 CPU, such as the A3000, but BASIC programs compiled with the HiSoft BASIC Compiler will do so and can be made to multitask as given here. Exec does all the hard work, really. All you have to do is learn to SLEEP while doing nothing. ■

Robert D'Asto is a Distribution Manager for Executive Software and a regular contributer to Amiga publications. Contact him c/o The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458.*

SAVE $35.75

A source of technical information for the serious Amiga professional.

Introducing *The AmigaWorld Tech Journal*, the new source to turn to for the advanced technical information you crave.

Whether you're a programmer or a developer of software or hardware, you simply can't find a more useful publication than this. Each big, bi-monthly issue is packed with fresh, authoritative strategies to help you fuel the power of your computing.

Trying to get better results from your BASIC compiler? Looking for good Public Domain programming tools on the networks and bulletin boards? Like to keep current on Commodore's new standards? Want to dig deeper into your operating system and even write your own libraries? Then *The AmigaWorld Tech Journal* is for you!

Our authors are programmers themselves, seasoned professionals who rank among the Amiga community's foremost experts. You'll benefit from their knowledge and insight on C, BASIC, Assembly, Modula-2, ARexx and the operating system—in addition to advanced video, MIDI, speech and lots more.

Sure, other programming publications may include some technical information, but none devote every single page to heavyweight techniques, hard-core tutorials, invaluable reviews, listings and utilities as we do.

Every issue includes a valuable companion disk!

And only *The AmigaWorld Tech Journal* boasts a technical advisory board composed of industry peers. Indeed, our articles undergo a scrupulous editing and screening process. So you can rest assured our contents are not only accurate, but completely up-to-date as well.

PLUS! Each issue comes with a valuable companion disk, including executable code, source code and the required libraries for all our program examples—plus the recommended PD utilities, demos of new commercial tools and other helpful surprises. These disks will save you the time, money and hassle of downloading PD utilities, typing in exhaustive listings, tracking down errors or making phone calls to on-line networks.

In every issue of *The AmigaWorld Tech Journal*, you'll find...

- Practical hardware and software reviews, including detailed comparisons, benchmark results and specs.
- Step-by-step, high-end tutorials on such topics as porting your work to 2.0, debugging, using SMPTE time code, etc.
- The latest in graphics programming, featuring algorithms and techniques for texture mapping, hidden-line removal and more.
- TNT (tips, news and tools), a column covering commercial software, books and talk on the networks.
- Programming utilities from PD disks, bulletin board systems and networks.
- Wise buys in new products—from language system upgrades to accelerator boards to editing systems and more.

The fact is, there's no other publication like *The AmigaWorld Tech Journal* available. It's all the tips and techniques you need. All in one single source. So subscribe now and get the most out of your Amiga programming. Get six fact-filled issues. And six jam-packed disks.
Call 1-800-343-0728 or complete and return the savings form below—today!

The AmigaWorld
TECH JOURNAL

To order, use this handy savings form.

☐ **Yes!** Enter my one-year (6 issues, plus 6 invaluable disks) Subscription to *The AmigaWorld Tech Journal* for just $59.95. That's a special saving of $35.75 off the single-copy price. If at any time I'm not satisfied with *The AmigaWorld Tech Journal*, I'm entitled to receive a full refund — no questions asked!

Name _____

Address _____

City _____ State _____ Zip _____

☐ Check or money order enclosed. ☐ Charge my:
☐ MasterCard ☐ Visa ☐ Discover ☐ American Express

Account No. _____ Exp. Date _____

Signature _____

Satisfaction Guaranteed!
Or your money back!

Canada and Mexico, $74.95.
Foreign surface, $84.97.
Foreign airmail, $99.95.
Payment required in U.S. funds drawn on U.S. bank.

Complete and mail to:
The AmigaWorld Tech Journal
P.O. Box 802, 80 Elm Street
Peterborough, NH 03458

TJ1091S

For faster service, call toll-free 1-800-343-0728.

LETTERS

Flames, suggestions, and cheers from readers.

ONE FOR THE OTHER SIDE

A follow-up article to Scott Hood's "The NTSC/RS-170A Standard" (p. 30, June/July '91) that details the PAL system would be particularly useful to U.S. and Canadian programmers.

Don Cox
Middlesbrough, Cleveland, England

Good idea. Scott is working on a sequel for the next issue, but it is about designing an NTSC-standard device for the Amiga. We'll see what we can do about PAL. You will be pleased, however, when you read our up-coming article "Programming for Both Sides of the Ocean."

PERFECTING PERFECTION

Rhett Anderson's "In Search of...The Perfect Joystick Routine" (p. 34, April/May '91) inspired me to take a look at my own joystick routine. I believe I am closer to perfection:

```
;joystick-move routine
;(on exit a0 points to change in x,y)
move.w  $dff00c,d0   ;joy1dat (read joystick)
ror.b       #2,d0   ;group bits together
lsr.w       #4,d0   ;shift right
and.w     #$3c,d0   ;clear all bits except 2,3,4,5
;table for change in x,y position
lea         joytable,a0
;a0 points to change in hor, vert postion
adda.w    d0,a0
rts
;these are pairs of x,y
joytable:  dc.w  0,0,0,1,1,1,1,0,0,–1,0,0,0,0
           dc.w  1,–1,–1,–1,0,0,0,0,0,0,–1,0,–1,1
```

My assembled code is only 24 bytes long. Compare this to Mr. Anderson's code at 36 bytes. Although my table is 54 bytes compared to his 50 bytes, my code is faster and smaller. I confess my code only checks one joystick. But, speed is of the essence, is it not?

Michael Morey
Portland, Oregon

MORE MIDI

In my quest for a programmable MIDI sequencer for the Amiga, I was quite impressed by Dan Babcock's article, "The Amiga Zen Timer" (p. 38, April/May '91). I had hoped to use its code to "time stamp" incoming MIDI data, but the Zen Timer is for very short (20-millisecond) events only.

I need the rudimentary source code, C or assembly, for a program that can receive, time stamp, and send MIDI data through an my A1000's MIDI interface. I am designing an interactive MIDI performance system. For comparison see the IBM programs in Jim Conger's book *C Programming for MIDI*. Are you planning any articles on designing a MIDI sequencer?

Ferdinand Kuhn
Alamogordo, New Mexico

Yes. The next issue has an article on the MIDI hardware spec. More will follow.

SHARE YOUR THOUGHTS

What are your suggestions, complaints, and hints? Write to Letters to the Editor, The AmigaWorld Tech Journal, *80 Elm St., Peterborough, NH 03458, or post messages in AW.TechJournal on BIX. Both may be edited for space and clarity.* ■

ANSWER: The test on page 50 prints Y O U G O T I T !

With some added comments, you should get:

```
int tryit(val, stringp)
int *val;  /* This is flag value we toggle */
           /* each time.  It is passed by */
           /* address so we can change it. */
char **stringp;  /* This is a pointer to a */
                 /* pointer to a character. Or */
                 /* as we are treating it, a */
                 /* pointer to an ARRAY of */
                 /* characters.  */
{
   (*stringp)++;  /* Skip to next character */
                  /* in the input string. Note: */
                  /* we always skip the first */
                  /* character.  */
   if (!**stringp) return(0);  /* When we hit */
           /* the end of the string, drop out. */
   (**stringp)- -;  /* Decrement the value of */
           /* the character we're looking at. */
           /* This changes B to A, C to B, ! */
           /* to space, etc. */
   *val = !*val;  /* Toggle flag. Each time */
                  /* we call this routine it will */
          /* reverse the setting of this value */
          /* (assuming they continue to pass */
          /* us the same value each time).  */
   if (*val) return(**stringp);  /* If an odd */
          /* invocation of this routine, return */
          /* the character we found.  */
   return(' ');  /* Otherwise return a space. */
}

void main()
{
   char b[] =
            "TZHPYVC!OHKLORYCISKCKX";
   /*   *Y*O*U* *G- - - - - - - - - - - - - */
   /* The * characters are skipped while */
   /* the others are decremented and */
   /* returned. The - characters are */
   /* overwritten below*/
   int j,  /* Toggle flag for the tryit routine */
     /* to track which characters to skip. */
   c;  /* The next character returned.  */
   char *p;   /* The pointer that the trying */
              /* routine will manipulate. */
   p = b;     /* Start out pointing at the */
              /* beginning of the buffer. */
   /* Note that assigning an array to the */
   /* pointer really assigns the address of */
   /* the base of the array to the pointer.  */
   strcpy(p+10, "RPNUS!EJFUL\"G");
   /*   *O*T*  *I*T*! *  These overwrite the - */
   /* characters in the b array */
   j = !p[0];  /* We treat p as a pointer to an */
          /* array and then take the logical */
      /* negation of that character. As it is */
      /* definitely non-zero,  this has the */
      /* effect of setting j to zero. */

   /* This loop just outputs each character */
   /* as it is encountered. */
   while(c = tryit(&j, &p))
      putchar(c);
   putchar('\n');
}
```

Free Product Info

DIRECT TO YOU
AT NO COST OR OBLIGATION

Want to Know More About Products Or Services Advertised In This Issue?

Here's How.

- Print your name and address where indicated.
- Tell us about yourself by answering the questions.
- Circle the numbers on the card corresponding to the reader service number on the ads for products or services that interest you.
- Tear out and mail the card.

NO OBLIGATION. Literature on products and services will be sent to you directly from advertisers, free of charge.

THE *AMIGAWORLD* TECH JOURNAL
READER SERVICE CARD

Name _____ Phone ( ) _____

Address _____

City _____ State _____ Zip _____

October, Issue Card
Valid Until January 31, 1992

1. Which of the following computer(s) do you presently own?
   a. ☐ Amiga 500
   b. ☐ Amiga 1000
   c. ☐ Amiga 2000
   d. ☐ Amiga 2500
   e. ☐ Amiga 3000
   f. ☐ Don't own a computer
   g. ☐ Other computer
2. What level of programmer/technician are you?
   h. ☐ Beginner
   i. ☐ Intermediate
   j. ☐ Advanced
   k. ☐ I am a developer
3. Would you like reviews of public domain software in the Reviews section?
   l. ☐ Yes m. ☐ No
4. Would you like to see articles in the *Tech Journal* covering:
   n. ☐ The technical aspects of video
   o. ☐ Hardware construction projects
   p. ☐ MIDI
5. Which language would you like to receive the most coverage in the *Tech Journal*? (Please chose one.)
   q. ☐ ARexx u. ☐ C++
   r. ☐ Assembly v. ☐ Fortran
   s. ☐ BASIC w. ☐ Modula-2
   t. ☐ C x. ☐ Pascal

To order a one year subscription to the *AmigaWorld Tech Journal* call 800-343-0728 or 603-924-0100 for immediate service. A one year subscription consists of 6 bi-monthly issues accompanied by 6 disks for a charter price of $59.95.

1	21	41	61	81
2	22	42	62	82
3	23	43	63	83
4	24	44	64	84
5	25	45	65	85
6	26	46	66	86
7	27	47	67	87
8	28	48	68	88
9	29	49	69	89
10	30	50	70	90
11	31	51	71	91
12	32	52	72	92
13	33	53	73	93
14	34	54	74	94
15	35	55	75	95
16	36	56	76	96
17	37	57	77	97
18	38	58	78	98
19	39	59	79	99
20	40	60	80	100

THE *AMIGAWORLD* TECH JOURNAL
READER SERVICE CARD

Name _____ Phone ( ) _____

Address _____

City _____ State _____ Zip _____

October, Issue Card
Valid Until January 31, 1992

1. Which of the following computer(s) do you presently own?
   a. ☐ Amiga 500
   b. ☐ Amiga 1000
   c. ☐ Amiga 2000
   d. ☐ Amiga 2500
   e. ☐ Amiga 3000
   f. ☐ Don't own a computer
   g. ☐ Other computer
2. What level of programmer/technician are you?
   h. ☐ Beginner
   i. ☐ Intermediate
   j. ☐ Advanced
   k. ☐ I am a developer
3. Would you like reviews of public domain software in the Reviews section?
   l. ☐ Yes m. ☐ No
4. Would you like to see articles in the *Tech Journal* covering:
   n. ☐ The technical aspects of video
   o. ☐ Hardware construction projects
   p. ☐ MIDI
5. Which language would you like to receive the most coverage in the *Tech Journal*? (Please chose one.)
   q. ☐ ARexx u. ☐ C++
   r. ☐ Assembly v. ☐ Fortran
   s. ☐ BASIC w. ☐ Modula-2
   t. ☐ C x. ☐ Pascal

To order a one year subscription to the *AmigaWorld Tech Journal* call 800-343-0728 or 603-924-0100 for immediate service. A one year subscription consists of 6 bi-monthly issues accompanied by 6 disks for a charter price of $59.95.

1	21	41	61	81
2	22	42	62	82
3	23	43	63	83
4	24	44	64	84
5	25	45	65	85
6	26	46	66	86
7	27	47	67	87
8	28	48	68	88
9	29	49	69	89
10	30	50	70	90
11	31	51	71	91
12	32	52	72	92
13	33	53	73	93
14	34	54	74	94
15	35	55	75	95
16	36	56	76	96
17	37	57	77	97
18	38	58	78	98
19	39	59	79	99
20	40	60	80	100

Free Product Info

DIRECT TO YOU
AT NO COST OR OBLIGATION

Please Use First Class Letter Postage

AmigaWorld TECH JOURNAL
P.O. Box 8751
Boulder, CO 80329-8751

Please Use First Class Letter Postage

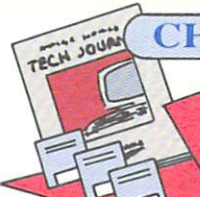
AmigaWorld TECH JOURNAL
P.O. Box 8751
Boulder, CO 80329-8751

Want to Know More About Products Or Services Advertised In This Issue?

Here's How.

- Print your name and address where indicated.
- Tell us about yourself by answering the questions.
- Circle the numbers on the card corresponding to the reader service number on the ads for products or services that interest you.
- Tear out and mail the card.

NO OBLIGATION. Literature on products and services will be sent to you directly from advertisers, free of charge.



CHARTER SUBSCRIPTION OFFER

AVAILABLE ONLY FROM AMIGA WORLD

SUBSCRIBE & SAVE

THE AMIGAWORLD TECH JOURNAL

YES! enter my one-year (6 bi-monthly issues, plus 6 invaluable disks) Charter Subscription to The AmigaWorld Tech Journal for the special price of $59.95. That's a savings of $35.75 off the single copy price. If at any time I'm not satisfied with The AmigaWorld Tech Journal, I'm entitled to receive a full refund — no questions asked.

Canada and Mexico $74.95. Foreign Surface $84.95. Foreign Airmail $99.95. Payment required in U.S. funds drawn on U.S. bank. Available March 15.

Name _____

Address _____

City _____ State _____ Zip _____

☐ Check/Money order enclosed TS411

Charge my: ☐ Mastercard ☐ AMEX ☐ Visa ☐ Discover

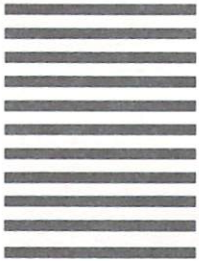
Account# _____ Exp. _____

Signature _____

Or call 800-343-0728 ☎ 603-924-0100 for immediate service.

NEW! 6 BI-MONTHLY ISSUES PLUS 6 INVALUABLE DISKS

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



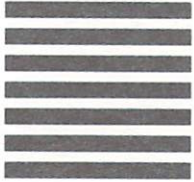
BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT #73 PETERBOROUGH, NH

POSTAGE WILL BE PAID BY ADDRESSEE

The AmigaWorld Tech Journal
P.O. Box 802
Peterborough, NH 03458-9971

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED
STATES

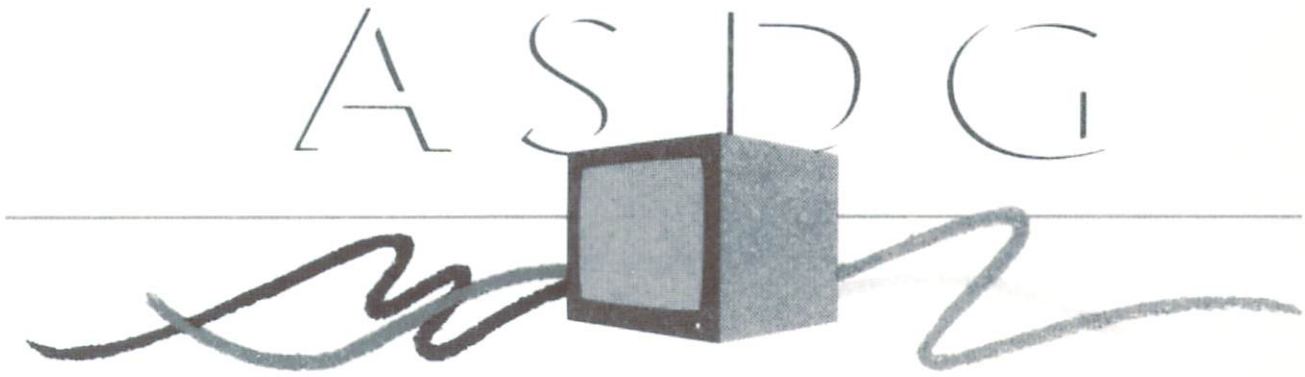


BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 73 PETERBOROUGH, NH

POSTAGE WILL BE PAID BY ADDRESSEE

AmigaWorld Tech Journal
P.O. Box 802
Peterborough, NH 03458-9971



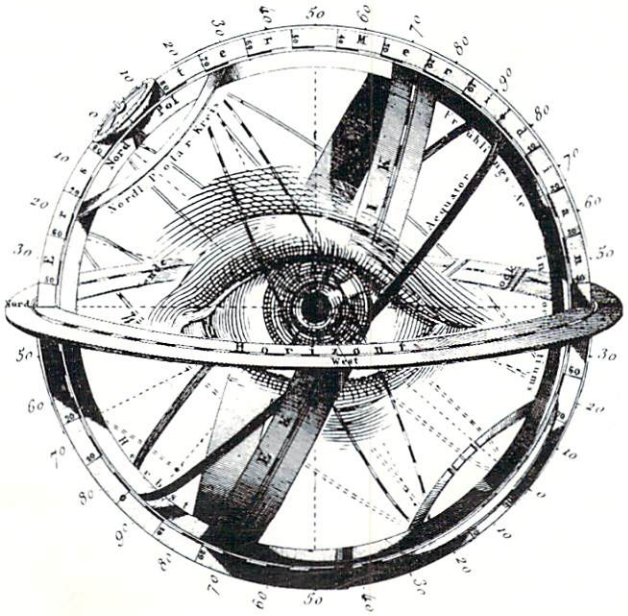
MEANS COLOR!

AT ASDG...
OUR ONLY BUSINESS IS COLOR.

If you've produced a quality color imaging product...
You can choose no better publisher than ASDG.

CALL US AT (608) 273 - 6585

SAY REVOLUTION



The fastest growing video technology company in the world is looking for programmers to join our team. We've assembled the hottest development group in the industry here at NewTek. But we have two slots open for a new project that will blow your socks off. Have you always dreamed of working on revolutionary technology in a small, focused group? We need software innovators that want to create the products of tomorrow. Here are the skills you'll need:

- Strong 68xxx assembly-language programming skills
- At least 3 years of assembly-language programming experience
- Ability to write low-level code for time-critical applications
- Background in high-speed graphics and video applications
- Experience in programming prototype hardware
- Ability to quickly learn new custom chip architectures
- Background in low-level I/O and interrupt operations
- Intimate understanding of Amiga O/S and hardware
- Experience with video and graphics hardware
- Ability to read a schematic
- Strong organizational and project design skills
- Being a self-motivated, self-teaching, innovator
- An uncompromising drive for excellence

If you've got what it takes, you'll be forging ahead where no programmer's ever gone before. NewTek offers outstanding (and unusual), compensation and benefit packages for the chosen few who are a cut above. You'll work in an environment created by hackers, designed to be a hackers heaven. Wouldn't it be fun to invent things that are featured in USA Today, Rolling Stone, and TIME? At NewTek your brain can change the world.

Send Resumes to:

Alcatraz
C/O NewTek
215 SE 8th St.
Topeka, KS 66603

NEWTEK
INCORPORATED

Circle 4 on Reader Service card.